November 1979

This document describes how to compile, link, and execute 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-H485A-TE

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V1.6
SOFTWARE VERSION:	VAX-11 PASCAL V1.0-1

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

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

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

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

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

The following are trademarks of Digital Equipment Corporation:

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

CONTENTS

CONTENTS

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

The VAX-11 PASCAL User's Guide is intended for use in developing 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 execution commands.
- Chapter 5 provides information about PASCAL input/output, including details on the use of logical names, file conventions, and record structure.
- Chapter 6 discusses the conventions followed in calling procedures, especially the conventions for passing parameters.
- Chapter 7 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 8 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.

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

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

CONVENTIONS USED IN THIS DOCUMENT

This document uses the following conventions.

Convention	Meaning
Uppercase words and letters	Uppercase words and letters, used in examples, indicate that you should type the word or letter exactly as shown.
Lowercase words and letters	Lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice.
[[ ]]	Double brackets indicate optional elements.
[ ]	Square brackets indicate that you must type the bracket characters.
{ }	Braces are used to enclose lists from which one element is to be chosen.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated one or more times.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.

Convention	Meaning
$ 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.

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, and executing 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 to the VAX/VMS operating system. These commands are:

```
$ EDIT file-spec
$ PASCAL file-spec
$ LINK file-spec
$ RUN file-spec
```

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.

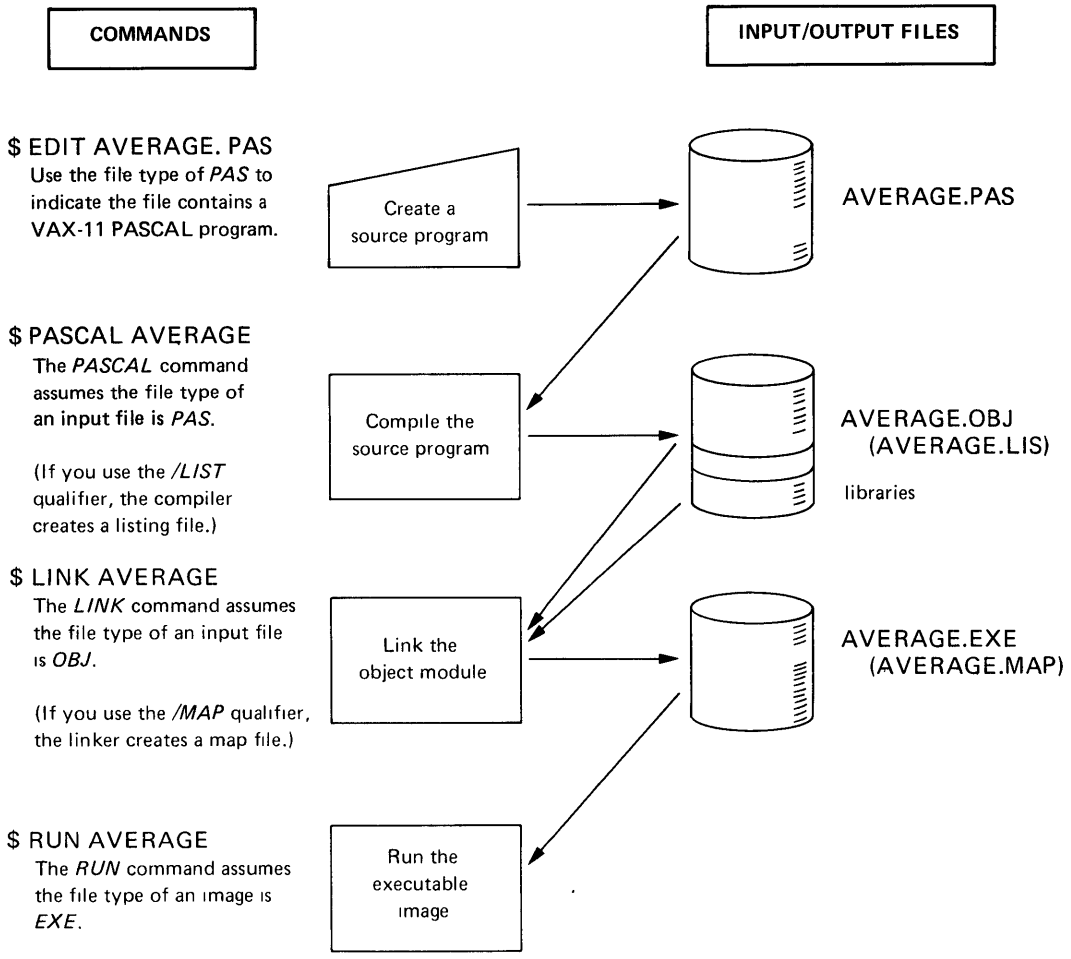


Figure 1-1 Program Development Process

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.filetype;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.

filetype

Describes the kind of data in the file; filetype 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 filetype 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 version 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 wish to use.

At your option, the compiler produces one or more object files, which are input to the linker (see Chapter 3), and one or more listing files. The listing files contain source 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[/qualifiers] file-spec-list[/qualifiers]
```

/qualifiers

Indicate special processing to be performed by the compiler.

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.

In interactive mode, you can also enter the file specification on a separate line by typing a carriage return 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 PASCAL compiler qualifiers to specify special processing.

Table 2-1 lists the qualifiers you can use with the VAX-11 PASCAL compiler. You can specify the qualifiers on the command line or 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 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 illegal 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 some DEBUG 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, DEBUG is disabled.

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 error 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]
```

You can include a file specification for the listing file. The default file specification designates 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 at each place where 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. Appendix C of the VAX-11 PASCAL Language Reference Manual lists 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 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 command is unique. For example, you can truncate CROSS_REFERENCE to CR, CHECK to CH, and DEBUG to D.

When you use PASCAL command qualifiers in command procedure files, it is recommended that you use the full qualifier names 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 options 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 type 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 must disable the LIST 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 AAA,BBB,CCC
```

   Source files AAA.PAS, BBB.PAS, and CCC.PAS are compiled as separate files, producing object files named AAA.OBJ, BBB.OBJ, and CCC.OBJ; and listing files named AAA.LIS, BBB.LIS, and CCC.LIS.

2.
```
$ PASCAL XXX+YYY+ZZZ
```

   Source files XXX.PAS, YYY.PAS, and ZZZ.PAS are concatenated and compiled as one file, producing an object file named XXX.OBJ. In batch mode, this command also produces the listing file XXX.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 AAA+BBB,CCC/LIST
```

   Two object files are produced: AAA.OBJ (comprising AAA.PAS and BBB.PAS) and CCC.OBJ (comprising CCC.PAS). In interactive mode, this command produces the listing file CCC.LIS. In batch mode, it produces two listing files: AAA.LIS and CCC.LIS.

5. $ PASCAL ABC+CIRC/NOOBJECT+XYZ

   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 ABC.PAS, CIRC.PAS, and XYZ.PAS will be concatenated and compiled, but no object file will be produced.

6. $ PASCAL/LIST [DIR]MNP

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

2.4 COMPILER LISTING FORMAT

When you specify the LIST qualifier, VAX-11 PASCAL produces a compiler listing. This section explains the format of the compiler listing illustrated in Figure 2-1.

```
EXAMPLE          11-OCT-1979   11:35:18   VAX-11 PASCAL VERSION  V1.0-1       PAGE    1
01               11-OCT-1979   11:34:47   DB2:[200,200]EXAMPLE.PAS;4 (1)

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

    100     1   1       program EXAMPLE(INPUT,OUTPUT);
    200     2   1
    300     3   1       label 10;
    400     4   1       var A,B,C:REAL;
    500     5   1
    600     6   1       begin
    700     7   1
    800     8       0   repeat
    900     9       2       WRITELN('Enter triangle sides');
   1000    10       2       if EOLN(INPUT) then goto 10;
   1100    11       2       READLN(A,B);
   1200    12       2       C := (SQR(A) + SQR(B)) ** 0.5;
%PAS-W-DIAGN                                    ^450                          ***    12 ==>     0
           *** WARNING 450: Nonstandard Pascal: Exponentiation
   1300    13       2       WRITELN('Hypotenuse is: ',C);
   1400    14       2   until FALSE;
   1500    15       1
   1600    16       1   10:
   1700    17       1   WRITELN('Done');
%PAS-F-DIAGN                                    ^20,*,4                       ***    17 ==>    12
           *** ERROR   4:  ")" expected
           *** ERROR  20:  "," expected
   1800    18       1
   1900    19       1   end.
           20       0

Compilation time =     0.56 seconds (  2036 lines/minute).

 2 Errors  1 Nonstandard feature
Last error(warning) on line     17.

Active options at end of compilation:

NODEBUG,STANDARD,LIST,NOCHECK,WARNINGS,CROSS_REFERENCE,
MACHINE_CODE,OBJECT,ERROR_LIMIT = 30
```

Figure 2-1 Compiler Listing Format

```
EXAMPLE         11-OCT-1979   11:35:18   VAX-11 PASCAL VERSION  V1.0-1       PAGE   2
MACHINE_CODE

GENERATED CODE (PRIOR TO BRANCH OPTIMIZATION)
 LINE  ADDRESS        OPCODE OPERANDS                          BYTESTREAM (HEXADECIMAL; READ FROM RIGHT TO LEFT)

          0002        MOVAB  *VAR(0, 0),R11                    5B 00 00 00 00 00 9E
          0009        MOVL   R13,*VAR(0, 4)                    00 00 00 00 00 5D D0
          0010        PUSHL  #0                                00 DD
          0012        CLRD   -(R14)                            7E 7C
          0014        PUSHL  R11                               5B DD
          0016        PUSHL  #67436548                         04 05 00 04 8F DD
          001C        CLRD   -(R14)                            7E 7C
          001E        PUSHL  #0                                00 DD
          0020        PUSHAL (R13)B^-20                        EC AD DF
          0023        CLRD   -(R14)                            7E 7C
          0025        PUSHL  #0                                00 DD
          0027        CALLS  #7,SYS$GETJPI (22)                00 00 00 00 00 07 FB
          002E        ADDL2  #8,R14                            5E 08 C0
          0031        ADDL2  #5636,(R11)                       6B 00 00 16 04 8F C0
          0038        PUSHAL (R11)B^8                          08 AB DF
          003B        CALLS  #1,PAS$INPUT                      00 00 00 00 00 01 FB
          0042 (24)   PUSHAL (R11)B^8                    (25)  08 AB DF
          0045        PUSHAL (R11)W^236                        00 EC CB DF
          0049        CALLS  #2,PAS$OUTPUT                     00 00 00 00 00 02 FB
(23) 8    0050        MOVL   R14,(R13)B^-12                    F4 AD 5E D0
     9    0054        MOVAB  (R11)W^236,R10                    5A 00 EC CB 9E
          0059        SUBL2  #16,R14                           5E 10 C2
          005C        PUSHL  R10                               5A DD
          005E        MOVAB  *VAR(2, 0),R9                     59 00 00 00 00 00 9E
          0065        MOVL   R9,(R14)B^4                       04 AE 59 D0
          0069        MOVL   #21,(R14)B^12                     0C AE 15 D0
          006D        MOVL   #21,(R14)B^8                      08 AE 15 D0
          0071        CALLS  #5,PAS$WRITESTR                   00 00 00 00 00 05 FB
          0078        MOVAB  (R11)W^236,R10                    5A 00 EC CB 9E
                        .
                        .
                        .

EXAMPLE         11-OCT-1979   11:35:18   VAX-11 PASCAL VERSION V1.0-1       PAGE   4
CROSS_REFERENCE

CROSSREFERENCE LISTING

 A                          4     11      12
 B (28)                     4     11      12 (26)
 C                    (29)  4     12      13
 INPUT                      1     10
 OUTPUT                     1

GLOBALLY DEFINED IDENTIFIERS:

 EOLN                       0     10
 FALSE                      0     14
 READLN                     0     11            (27)
 REAL                       0      4
 SQR                        0     12      12
 WRITELN                    0      9      13      17
```

Figure 2-1 (Cont.) Compiler Listing Format

The compiler listing contains the following three sections:

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

The numbers throughout this section are keyed to the numbers in Figure 2-1.

Title Line -- Each page of the listing contains a title line. The title line lists the module name (1), the date and time of the compilation (2), the PASCAL compiler name and version number (3), and the listing page number (4).

2.4.1 Source Code Listing

Each page of the source code listing contains a line under the title line specifying the date and time of source file creation (5) and the VAX/VMS file specification of the source file (6).

Source Code Listing -- The lines of the source code are printed in the source code listing. In addition, the listing contains the following information pertaining to the source code:

- SOS line numbers (7) -- 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.

- Line numbers (8) -- 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.

- Procedure level (9) -- 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.

- Statement level (10) -- 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.

Errors and Warnings -- 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. In addition, the error description can contain the following information:

- A circumflex (^) that points to the character position in the line where the error was detected (11).
- A numeric code, following the circumflex, that specifies the particular error (12). On the following lines of the source listing, the compiler prints the text that corresponds to each numeric code (13). Note that one source program error often causes the PASCAL compiler to detect more than one error (14).

- An asterisk (*) that shows where the compiler resumed translation after the error (15).

- The line number in which the error was detected (16) and the line number of the last line containing an error diagnostic (17). You can use these error line numbers to trace the error diagnostics backwards through the source listing.

Summary -- At the end of the source listing, the compiler tells you how much time was required for the compilation (18). If your program generated warning or error messages, the compiler prints a summary of all the errors (20) and the source line number of the last message (19). Finally, the compiler lists the status of all the compilation options (21).

2.4.2 Machine Code Listing

The machine code listing (if requested with the MACHINE_CODE qualifier) follows the source listing. The machine code listing contains:

- Symbolic representation (22) -- The symbolic representation, similar to a VAX-11 MACRO instruction, appears for each object instruction generated. Because the PASCAL compiler operates in one pass, it must generate these instructions before it performs branch optimization. Branch optimization can cause certain BRW instructions to be deleted. Therefore, these instructions will not be identical to those appearing in the executable image.

- Source line number (23) -- A source line number marks the first object instruction that the compiler generated for the first PASCAL statement on that source line.

- Hexadecimal address (24) -- The hexadecimal address is an approximation of the address of the object instruction. You should not use these addresses for debugging purposes because they do not correctly correspond to the locations in the executable image. The branch optimization mentioned above can change the addresses of the object instructions.

- Hexadecimal instruction (25) -- This is the hexadecimal representation of the object instruction. You should read the hexadecimal instruction from right to left because the rightmost byte has the smallest address. Again, because of its one-pass operation, the compiler must generate some object instructions before it can determine the address bytes of their operands. The addresses of these operands are printed as zeros. After generating the hexadecimal representation of an instruction, but before writing the object code file, the compiler places the correct values into the binary object code.

2.4.3 Cross-Reference Listing

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

- User-specified identifiers (26) -- This section lists all the identifiers you declared.

- Globally-defined identifiers (27) -- This section lists the PASCAL predefined identifiers that the program uses.

Each line of the cross-reference listing contains an identifier (28) and a list of the source line numbers where the identifier is used (29). 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.

CHAPTER 3

LINKING A PROGRAM

After compiling your VAX-11 PASCAL program 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

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

```
$ LINK[/command-qualifier(s)] file-spec[/file-qualifier(s)...]
```

/command-qualifier(s)

Specify output file options.

file-spec

Specifies the input object file to be linked.

/file-qualifier(s)

Specify input file options.

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 you can type a hyphen (-) as the last character of the line and continue on the next line.

You can enter multiple file specifications separated from each other by commas or plus signs. When used with the LINK command, the comma has the same effect as the plus sign: no matter which you use, the linker creates a single executable image from the 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.

Table 3-1 lists the linker qualifiers of particular interest to PASCAL users. See the VAX-11 Linker Reference Manual for details on the linker.

Table 3-1
Linker Qualifiers

Type of Qualifier	Qualifier	Negative Form	Default
Command qualifiers	/BRIEF	None	Not applicable
	/CROSS_REFERENCE	/NOCROSS_REFERENCE	/NOCROSS_REFERENCE
	/DEBUG	/NODEBUG	/NODEBUG
	/EXECUTABLE[=file-spec]	/NOEXECUTABLE	/EXECUTABLE
	/FULL	None	Not applicable
	/MAP[=file-spec]	/NOMAP	/NOMAP (interactive) /MAP (batch)
	/SHAREABLE[=file-spec]	/NOSHAREABLE	/NOSHAREABLE
	/TRACEBACK	/NOTRACEBACK	/TRACEBACK
Input file qualifiers	/INCLUDE=module-name(s)	None	Not applicable
	/LIBRARY	None	Not applicable

3.2 LINKER COMMAND QUALIFIERS

LINK command qualifiers modify the output of the linker and specify whether the debugging or the traceback facility is to be included. Linker output consists of an image file and, optionally, a map file. The following qualifiers, described in Section 3.2.1, control the image file generated by the linker:

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

The map file qualifiers, described in Section 3.2.2, are:

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

The debugger and traceback qualifiers, described in Section 3.2.3, are:

```
/DEBUG
/TRACEBACK
```

3.2.1 Image File Qualifiers

The image file qualifiers include:

```
/EXECUTABLE
/SHAREABLE
```

If you do not specify an image file qualifier, the default is /EXECUTABLE; the linker produces an executable image. To suppress production of an image, specify the negative form, as:

```
/NOEXECUTABLE
```

For example:

```
$ LINK/NOEXECUTABLE CIRCLE
```

The file CIRCLE.OBJ is linked, but no image is generated. The /NOEXECUTABLE qualifier is useful if you want 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. It can be a private image you use for your own applications, or it can be installed in the system by the system manager for use by all users. To create a shareable image, specify /SHAREABLE. For example:

```
$ LINK/SHAREABLE CIRCLE
```

To include a shareable image as input to the linker, you must 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 you specify /NOSHAREABLE, the linker generates an executable image.

3.2.2 Map File Qualifiers

The map file qualifiers tell the linker whether to generate a map file and, if so, the information the map file is to include. The map file qualifiers are:

```
/MAP
/BRIEF
/FULL
/CROSS_REFERENCE
```

The map qualifiers are specified as follows:

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

Note that you must specify /MAP if you specify /BRIEF, /FULL, or /CROSS_REFERENCE.

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, the map file has the name of the first input file and a file type of MAP. It is stored on the default device, in the default directory.

The optional qualifiers /BRIEF and /FULL define the amount 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 you specify neither /BRIEF nor /FULL, 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 you specify /DEBUG at link time, the program always executes under the control of the debugger, unless you specify /NODEBUG with the RUN command. The default at link time is /NODEBUG.

When you specify the /TRACEBACK qualifier, error messages are accompanied by symbolic traceback information showing the sequence of calls that transferred control to the program unit in which the error occurred. If you specify /NOTRACEBACK, this information is not 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 LINKER INPUT FILE QUALIFIERS

Input file qualifiers 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
```

This 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)
```

This 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 you use both qualifiers on 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[/[NO]DEBUG]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 debugger, even if you omitted this qualifier from the PASCAL and LINK commands (see Sections 2.2 and 3.1). 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 none of these qualifiers at any point in the compile-link-execute sequence, and an execution error occurs, you will 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

To perform symbolic debugging, you must use the /DEBUG qualifier with both the PASCAL command and the LINK command. It then is unnecessary to specify it 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 will not contain debug records and symbol tables used in debugging, and you will be forced to express addresses as absolute values, rather than symbolically.

If you specify LINK/NOTRACEBACK, you will receive no traceback list in the event of 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 condition 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 (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 to you 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. To

display traceback information, you can use the debugger command SHOW CALLS, as described in Section 4.1.2. For more information on using the debugger with VAX-11 PASCAL programs, refer to the VAX-11 PASCAL Installation Guide/Release Notes.

```
$ PASCAL/LIST TRACE
$ TYPE TRACE
TRACETEST        3-OCT-1979   11:21:08   VAX-11 PASCAL VERSION V1.0-1          PAGE  1
01               3-OCT-1979   11:20:41   DB1:[TEST.PAS]TRACE.PAS;1 (1)

     LINE       LEVEL
    NUMBERS   PROC STMT STATEMENT

           1   1       PROGRAM TRACETEST;
           2   1
           3   2        PROCEDURE P1 (VAR X : REAL);
           4   2        BEGIN
           5        0     X := 1.0/X;
           6        1    END;
           7        0
           8   2        PROCEDURE P2 (Y : REAL);
           9   2        BEGIN
          10        0     P1(Y);
          11        1    END;
          12        0
          13        0  BEGIN
          14        0    P2(0.0);
          15        1  END.
          16        0
          17        0

Compilation time =     0.68 seconds (  1500 lines/minute).


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

$ LINK TRACE
$ RUN TRACE

%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
```

Figure 4-1 Source Program Listing Traceback List

4.1.2 SHOW CALLS Command

When an error occurs in a program that is executing under the control of the debugger, no traceback list is produced. To generate a traceback list, use the SHOW CALLS command, which has the form:

```
DBG>SHOW CALLS
```

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 1.6

$ 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 RET
$ LINK CIRCLE RET
$ RUN CIRCLE RET
```

CHAPTER 5

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

5.1 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 particular file specifications. Table 5-1 lists the logical names of special interest to PASCAL users.

In addition, the system manager defines the logical names PAS$INPUT and PAS$OUTPUT, which default to SYS$INPUT and SYS$OUTPUT, at compiler installation.

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 5-1
Predefined System Logical Names

Name	Meaning	Default
SYS$DISK	Default device and directory	As specified by the user
SYS$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', 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).

5.2 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.

5.2.1 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). As a result, records can be added only at the end of the file.

5.2.2 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).

5.3 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.

5.3.1 Fixed-Length Records

When you specify fixed-length records (see Section 5.4.4), 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.

5.3.2 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 two binary bytes on a disk device and four decimal digits on magnetic tape. The value stored in the count field indicates the number of data bytes in the record.

5.4 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 (RMS) affect VAX-11 PASCAL. For more information, refer to the VAX-11 Record Management Services Reference Manual.

The OPEN procedure has the following general format:

```
OPEN (file-variable, 'VAX/VMS file specification', buffer-size,
      file-status, record-access-mode, record-type,
      carriage-control);
```

Buffer size, file status, record access mode, record type, and carriage control are RMS-dependent attributes described in this section.

5.4.1 Buffer Size

The buffer size 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 file of any other type, this parameter has no meaning.

5.4.2 File Status

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

```
NEW
OLD
```

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

If you specify OLD, the system tries to open an existing file. 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.

5.4.3 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.

5.4.4 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.

5.4.5 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
NOCARRIAGE
```

A value of LIST indicates that each record will be 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.

A value of CARRIAGE indicates that the first byte of each record contains a carriage control character. CARRIAGE is equivalent to the VAX-11 RMS record attribute FTN.

A value of NOCARRIAGE indicates that the record contains no carriage control information. NOCARRIAGE is equivalent to the VAX-11 RMS record attribute PRN with all bits equal to zero.

5.5 LOCAL INTERPROCESS COMMUNICATION: MAILBOXES

It is often useful to exchange data between processes: 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$CREMBX) system service (see Section 6.2.1 for an example using SYS$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', OLD, 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.

5.6 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 UPDATE 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 (NETJOB, 'BOSTON::"TASK=UNA"',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',OLD);
        RESET (NEWDAT);
        OPEN (BRANCH,'NASHUA"PLUGH XYZZY"::MASTER.DAT',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 6

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.

6.1 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. The illustrations in Appendix B outline the events that occur during a procedure call and show the structure of the run-time stack after each event.

6.1.1 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.

6.1.2 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 subsections 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 6.1.5. Refer to the VAX-11 PASCAL Language Reference Manual for a description of parameter passing between PASCAL subprograms.

6.1.2.1 **By-Reference Mechanism** - The by-reference mechanism passes the address of the actual parameter. By default, PASCAL uses this mechanism for everything 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 invokes PASCAL by-value semantics.
2. By using the VAR specifier in the formal parameter list. This syntax invokes 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 must 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, in floating-point notation, the hyperbolic tangent of an angle. 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 6.3 for more information on calling Run-Time Library procedures.)

Use the VAR specifier to pass an actual parameter that can change in value during 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 BOOLEAN;

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.

6.1.2.2 **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. You can use %IMMED only with non-PASCAL subprograms.

When you call the procedure, 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 names, as described in Section 6.1.3.

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.

6.1.2.3 **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.)

6.1.3 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.

6.1.4 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

6.1.5 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.

6.2 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 6.1.2. 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
```

6.2.1 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 codes indicate successful completion and even 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.

6.2.2 Calling System Services as Procedures

If you do not need to check the return status, you can declare a system service as an external procedure rather than 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.

6.2.3 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 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.

6.2.3.1 **Input and Output By-Reference Parameters** - You will often 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 eight bits of the variable will not be overwritten on output and the variable will not contain the data you expect. Table 6-1 lists the suggested input and output variable types.

Table 6-1
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);
```

6.2.3.2 **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);
```

6.2.3.3 **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 POS_WORD = 0..65535;
     SUB63 = 1..63;
     WORD_TYPE = PACKED RECORD
                 SHORTWD : POS_WORD
                 END;
     STRING_BUF = PACKED ARRAY [1..128] OF CHAR;

VAR  ICODE : INTEGER;
     NAMLEN : WORD_TYPE;
     NAMDES : STRING_BUF;

     .
     .
     .
```

```
PROCEDURE ERROR;
.
.
.
FUNCTION SYS$TRNLOG (%STDESCR CYGNUS : PACKED ARRAY [SUB63] OF CHAR;
                     VAR RSLLEN : WORD_TYPE;
                     %STDESCR RSLBUF : STRING_BUF;
                     %IMMED TABLE, ACMODE, DSBMSK : INTEGER) : INTEGER; EXTERN;
BEGIN
.
.
.
     ICODE := SYS$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 value 4, causes the system to disable its search of the process logical name table; only the system and group tables are searched.

Section 6.4 presents another complete system service example.

6.3 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.

6.4 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
        set 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 I.D.
                PTR_PIDLEN -- Address of process I.D. 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 7

ERROR PROCESSING AND CONDITION HANDLERS

During the execution of your VAX-11 PASCAL program, various conditions, including errors and exceptions, can occur. These conditions can 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. Thus, you have two options in dealing with errors. You can allow the Run-Time Library to provide all condition handling for you. 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 7.1 describes error processing by the Run-Time Library as it applies to VAX-11 PASCAL. Appendix A describes the error messages that can be generated for a PASCAL program.

At the lowest level, the VAX-11 Condition Handling Facility provides all condition handling for the Run-Time Library. You can also use it directly to provide procedures (user-written condition handlers) for processing conditions that occur during your program's execution. The use of condition handlers, however, 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, and the VAX-11 Architecture Handbook before you attempt to write a condition handler.

7.1 RUN-TIME LIBRARY DEFAULT ERROR PROCESSING

By default, the Run-Time Library prints a message and terminates PASCAL program execution when an error occurs. These default actions occur unless your program includes a condition handler.

Run-time errors are reported in the following format:

```
%PAS-F-code, text
```

The code is an abbreviation of the error message text, which explains the error. VAX-11 PASCAL run-time errors and recovery procedures are described in Appendix A. Most Run-Time Library procedures provide their own error messages, as described in the VAX-11 Run-Time Library Reference Manual.

7.2 OVERVIEW OF VAX-11 CONDITION HANDLING

When the VAX/VMS system creates a user process, it establishes a system-defined condition handler which, in the absence of any user-written condition handler, processes errors that occur during execution of the user image. 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 a condition handler to respond. The system conducts the search by calling handlers in preceding stack frames until it activates a condition handler that does not resignal. The default handler calls the system's message output routine so that it can 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 permits, program execution continues. Otherwise, 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, you can create and display messages that specifically describe conditions encountered during execution of an application program, instead of relying on the standard system error messages.

7.2.1 Condition Signals

A condition signal consists of a call to one of the two system-supplied signal procedures, LIB$SIGNAL and LIB$STOP. The signal procedures must be declared external, for example:

```
PROCEDURE LIB$SIGNAL (%IMMED CONDITION : INTEGER); EXTERN;

PROCEDURE LIB$STOP (%IMMED CONDITION : INTEGER); EXTERN;
```

You signal a condition when you do not want to handle it in the routine in which it was detected, but want instead to pass notification to other active routines. If the current procedure can continue after the signal is made, call LIB$SIGNAL. A higher-level procedure can then determine whether program execution is to continue. If the condition will not allow the current procedure to continue, call LIB$STOP.

Condition values are usually expressed as condition symbols. For example:

```
LIB$SIGNAL (MTH$_FLOOVEMAT);
```

You can include additional parameters to provide supplementary information about the error.

When called, a signal procedure searches for condition handlers by examining the preceding stack frames until it activates a condition handler that does not resignal.

7.2.2 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 it can correct the condition. If it can, the handler takes the appropriate action, and execution continues. If it cannot correct the condition, the handler may resignal the condition. That is, it requests that another condition handler process the exception.

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. Among them are:

- Continuing from the signal. If the signal was issued through a call to LIB$STOP, the program will exit.
- 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 procedure.
- Unwinding to the establisher's caller (the procedure that called the procedure that established the handler). The handler can determine the function value returned by the called procedure.

7.3 WRITING A CONDITION HANDLER

The following sections describe how to code and establish a condition handler, and provide some simple examples. See Appendix C of 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.

7.3.1 Establishing and Removing Handlers

When a procedure is called, no condition handler is initially established. To use a condition handler, you must first define both the Run-Time Library procedure LIB$ESTABLISH and the condition handler as follows:

```
FUNCTION ERR_HANDLER : INTEGER;  EXTERN;
PROCEDURE LIB$ESTABLISH (%IMMED FUNCTION HANDLER : INTEGER);
          EXTERN;
```

To establish the handler, you must call LIB$ESTABLISH. To remove an established handler, define the Run-Time Library procedure LIB$REVERT, and call it as follows:

```
PROCEDURE LIB$REVERT; EXTERN;
.
.
.
LIB$REVERT;
```

As a result of this call, the condition handler established in the current procedure activation is removed. When a procedure returns, the condition handler established by the procedure 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.

7.3.2 Parameters for Condition Handlers

A PASCAL condition handler is an INTEGER function that is called when an exception condition occurs. You must define two formal VAR parameters for a condition handler:

1. An integer array to refer to the parameter list from the call to the signal procedure (the signal parameters). That is, the list of parameters included in calls to LIB$SIGNAL or LIB$STOP (see Section 7.2.1).

2. An integer array to refer to information concerning the procedure activation that established the condition handler (the mechanism arrays).

For example, you can define a condition handler as follows:

```
TYPE MECHARR = ARRAY[0..4] OF INTEGER;
     SIGARR = ARRAY [0..9] OF INTEGER;
.
.
.
FUNCTION ERR_HANDLER (VAR SIGARGS : SIGARR;
                      VAR MECHARGS : MECHARR) : INTEGER;
     BEGIN
     .
     .
     .
     END;
```

```
PROCEDURE LIB$ESTABLISH (%IMMED FUNCTION HANDLER : INTEGER); EXTERN;
.
.
.
BEGIN
.
.
.
LIB$ESTABLISH (ERR_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 7.3.4 for a discussion of condition values.
SIGARGS[2 to n]	Indicates optional parameters as specified by 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 procedure that established the condition handler. The values from this procedure are listed below.

Value	Meaning
MECHARGS[0]	Specifies the number of parameters in this array (4).
MECHARGS[1]	Contains the address of the procedure activation 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 procedure activation, up to the point at which the exception occurred.
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.

7.3.3 **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, however, 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 (%IMMED 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:

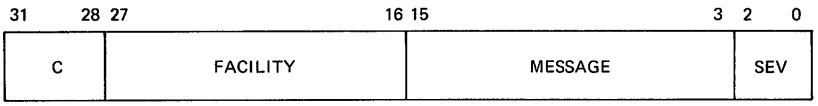
```
STATUS := SYS$UNWIND (MECHARGS[2],0);
```

- Unwind to the procedure that called the establisher. Specify:

```
STATUS := SYS$UNWIND (0,0);
```

7.3.4 Condition Values and Symbols

VAX-11 uses condition values to indicate that a called procedure 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. A condition value has the form:



Field	Bits	Meaning
C	31:28	Control bits
Facility	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.
Message	15:3	Describes the condition that occurred. Bit 15 = 1 indicates the message is specific to a single facility. Bit 15 = 0 indicates a system-wide message.

Field	Bits	Meaning
Sev	2:0	Specifies a severity code, as follows: 0 - warning 1 - success 2 - error 3 - information 4 - severe error 5, 6, 7 - reserved

A warning severity code (0) indicates that output was produced, but that the results might not be what you expected. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results will not all be correct. A severe error code (4) indicates that the error was of such severity that output was not produced. 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.

The condition value is passed as the second element of the array SIGARGS. Occasionally, your condition handler may require that 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 its 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, you may want to respond to a condition, regardless of the value of the severity code. 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...
```

7.3.5 Floating-Point Operation

Some conditions involving floating-point operations require special action if you want to continue execution. Operations that involve, for example, floating overflow, dividing by 0, or computing the square root of a negative number store 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, you must 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. Note that you should specify the third parameter as a double-precision variable. This ensures that the reserved operand will be changed correctly regardless of its precision. If you do not have a special value for this parameter, specify 0.0. For more information on LIB$FIXUP_FLT, see the VAX-11 Run-Time Library Reference Manual.

7.4 CONDITION HANDLER EXAMPLES

The example in this section illustrates how to declare and use a condition handler with a typical PASCAL procedure. It establishes a condition handler 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 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

And it declares these global variables:
        FLAG -- a Boolean 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$ESTABLISH (%IMMED FUNCTION FIXER : INTEGER); EXTERN;
        (*The system service procedure LIB$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 8

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

8.1 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 8-1 lists the possible program section attributes.

Table 8-1
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 8-2 summarizes the usage and attributes of these program sections.

Table 8-2
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.

8.2 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 8-3.

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 one 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 one byte and aligned on a byte boundary.

A pointer is simply a longword containing an address.

Table 8-3
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

8.3 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 a longword and aligned on a longword boundary. Similarly, each element of a character array is stored in a 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 8-1 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.

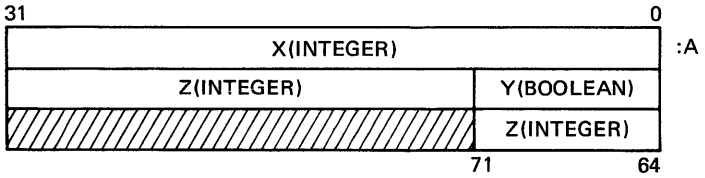


Figure 8-1 Storage of Sample Record

8.4 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 subsections 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.

8.4.1 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 one. 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 one, 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.

8.4.2 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 8-4 lists the space requirements for elements of packed arrays.

Table 8-4
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 SAMP1_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 SAMP1_ARR, in turn, requires 5 longwords -- enough storage space for 5 elements of type INTERNAL_ARR. The entire array SAMP1_ARR therefore occupies 25 longwords (800 bits).

2.
```
VAR SAMP1_ARR :  PACKED ARRAY [1..5,1..5] OF 0..6;
VAR SAMP1_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 SAMP1_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.

8.4.3 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 one 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 8-2 shows how this record is stored.

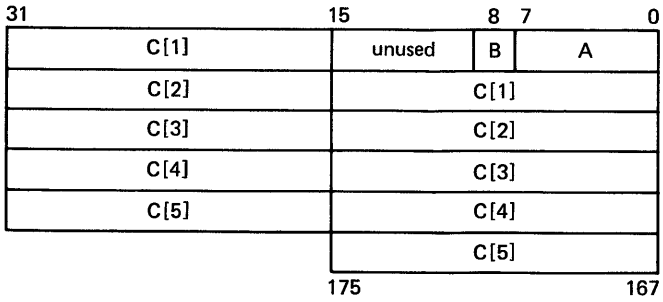


Figure 8-2 Storage of Sample Record

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 8-3 shows how this record is stored.

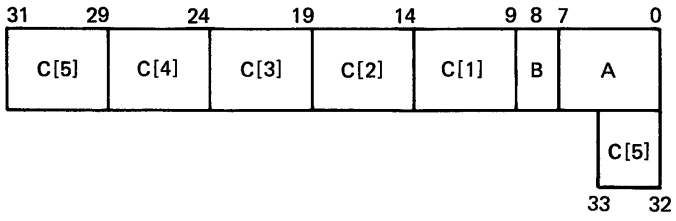


Figure 8-3 Storage of Sample Packed Record Containing Packed Array

This record requires a total of 34 bits of storage.

8.5 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.

8.5.1 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 8-4.

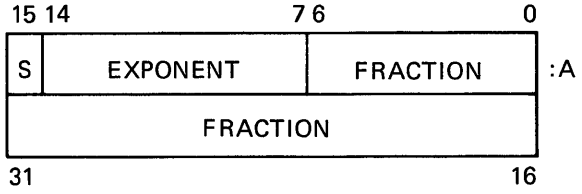


Figure 8-4 Single-Precision Floating-Point Data Representation

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.

8.5.2 **Double-Precision Floating-Point Data (DOUBLE Type)**

A double-precision floating-point value is represented by eight contiguous bytes. The bits are numbered from the right 0 through 63, as shown in Figure 8-5.

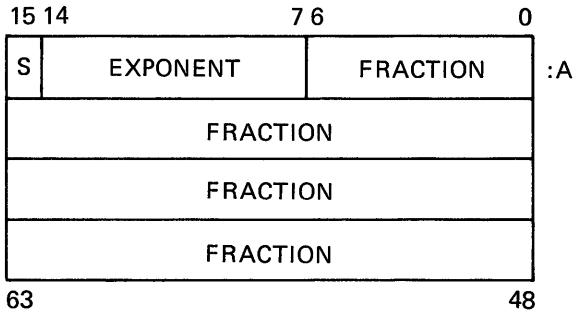


Figure 8-5 Double-Precision Floating-Point Data Representation

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-l-DIAGN
```

The l represents the severity level of the error. 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 .. 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 legal 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 may 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 may 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 may 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 may 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 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 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 is 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 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 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 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

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 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 PC = xxx

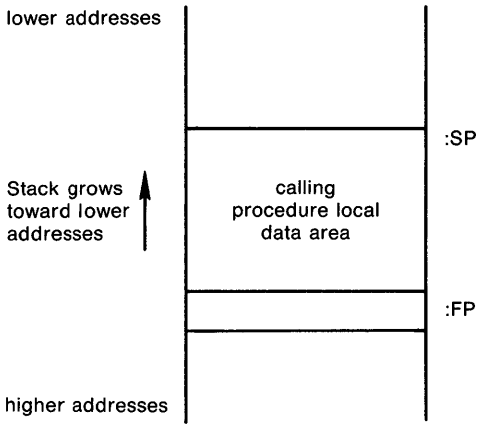
The program tried to assign an illegal value to a subrange variable. Make sure that all subrange assignments specify values that are within the bounds of the subrange. This message appears only if CHECK is enabled.

APPENDIX B

CONTENTS OF RUN-TIME STACK DURING PROCEDURE CALLS

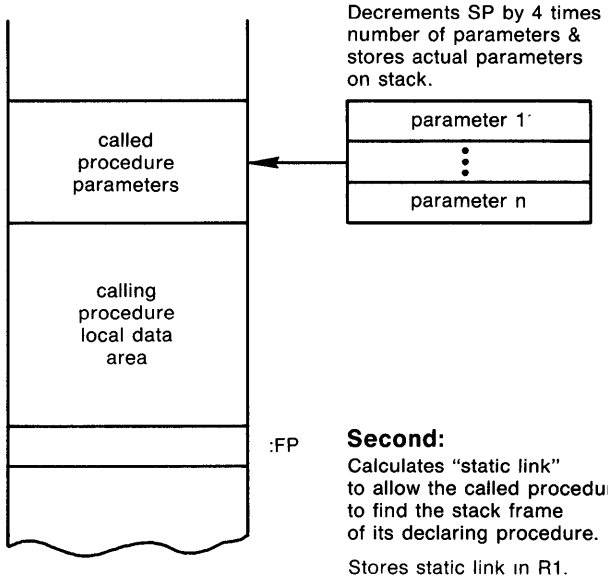
Figure B-1 outlines the events that occur during a procedure call and shows the contents of the run-time stack after each event.

1 Before procedure call:



SP = Stack Pointer
FP = Frame Pointer

2 The calling procedure's actions:



First:
Decrements SP by 4 times number of parameters & stores actual parameters on stack.

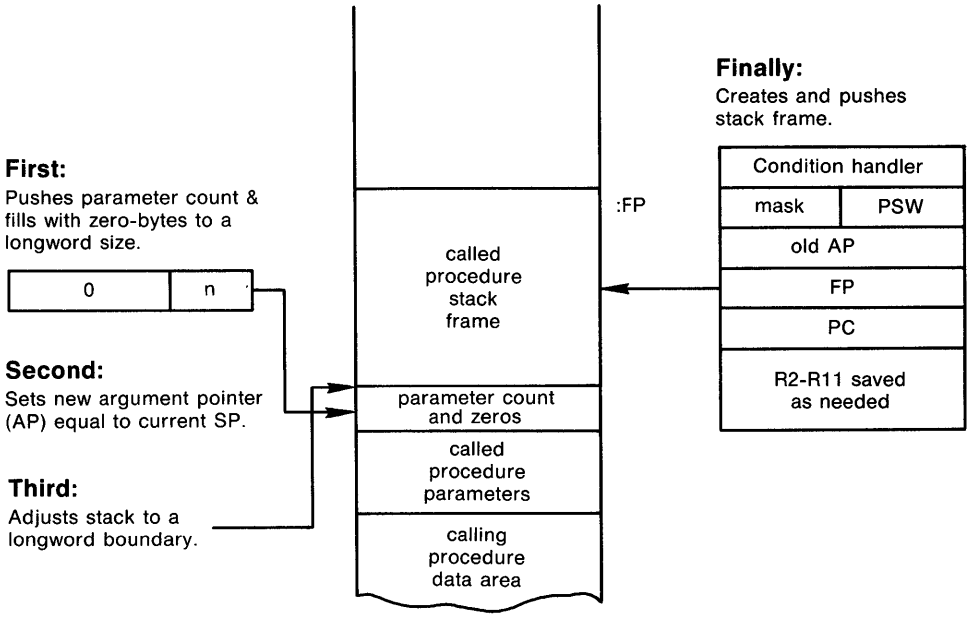
Second:
Calculates "static link" to allow the called procedure to find the stack frame of its declaring procedure.

Stores static link in R1.

Finally:
Issues CALLS instruction.

Figure B-1 Contents of Run-Time Stack During Procedure Calls

3 The CALLS instruction's actions:



4 The called procedure's actions:

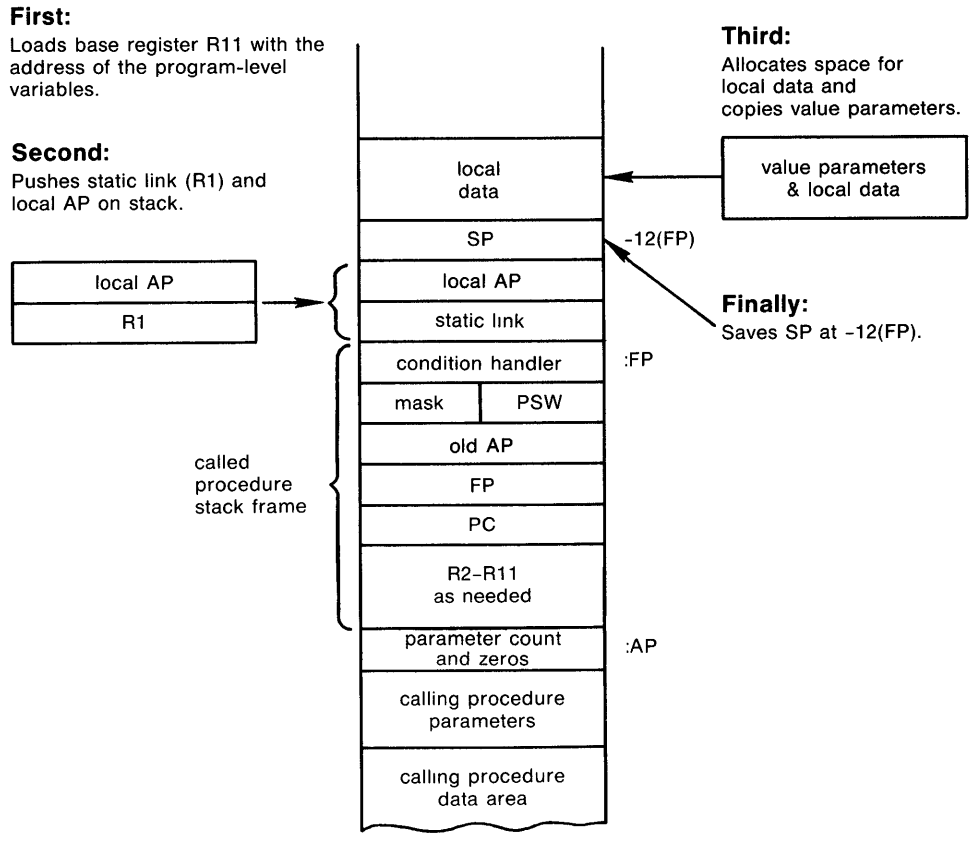


Figure B-1 (Cont.) Contents of Run-Time Stack During Procedure Calls

A

B

C

D

E

F

I

L

M

N

O

P

Q

R

S

T

U

V

W

Z

VAX-11 PASCAL
User's Guide
AA-H485A-TC

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

RT/C SOFTWARE PUBLICATIONS TW/A14
DIGITAL EQUIPMENT CORPORATION
1925 ANDOVER STREET
TEWKSBURY, MASSACHUSETTS 01876

— Do Not Tear - Fold Here —

Cut Along Dotted Line