VAX PASCAL User Manual

Order Number: AI–H485E–TE

February 1987

This manual is designed for programmers who have full working knowledge of VAX PASCAL. It describes how to interact with the VAX/VMS operating system using VAX PASCAL and how to use the features of VAX PASCAL to take full advantage of the VAX/VMS systems programming environment.

Revision/Update Information:	This revised document supersedes information in the *VAX PASCAL User's Guide* (Order Number AA–H485D–TE).
Operating System and Version:	VAX/VMS Version 4.4 or higher
Software Version:	VAX PASCAL Version 3.5

digital equipment corporation
maynard, massachusetts

First Printing, November 1979
Revised, March 1985
Revised, February 1987

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

ZK3231

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by TeX, the typesetting system developed by Donald E. Knuth at Stanford University. TeX is a trademark of the American Mathematical Society.

Contents

USING VAX PASCAL FEATURES ON THE VAX/VMS SYSTEM

APPENDIX F OPTIONAL PROGRAMMING PRODUCTIVITY TOOLS F-1

FIGURES

TABLES

Preface

Intended Audience

This manual is designed for programmers who have full working knowledge of VAX PASCAL. It describes how to interact with the VAX/VMS operating system using VAX PASCAL and how to use the features of VAX PASCAL to take full advantage of the VAX/VMS systems programming environment.

Structure of This Document

This manual consists of the following chapters and appendixes:

- Chapter 1 contains an overview of the VAX PASCAL language.
- Chapter 2 explains how to use the VAX/VMS operating system.
- Chapter 3 gives a description of various editors and provides information on how to compile, link, and execute VAX PASCAL programs.
- Chapter 4 provides information on debugging VAX PASCAL programs.
- Chapter 5 provides information on program section use, storage allocation, and data representations.
- Chapter 6 provides information on input and output operations.
- Chapter 7 discusses error processing, in particular, how to use the VAX/VMS condition-handling facility.
- Chapter 8 describes conventions followed in calling procedures.
- Chapter 9 explains optimizations performed by the VAX PASCAL compiler and techniques for writing efficient programs and modules.
- Appendix A lists complete run-time and compile-time error messages.
- Appendix B lists errors detected by the STATUS and STATUSV functions.

- Appendix C describes the entry points to utilities in the VAX/VMS Run-Time Library that can be called as external VAX PASCAL routines.
- Appendix D discusses the differences between VAX PASCAL Version 1 and all subsequent versions of VAX PASCAL.
- Appendix E contains examples of using system routines.
- Appendix F describes how to use two optional programming productivity tools: the VAX Language-Sensitive Editor (VAXLSE) and VAX Source Code Analyzer (VAXSCA).

Associated Documents

- The *VAX PASCAL Reference Manual* provides detailed language information.
- The *VAX PASCAL Installation Guide* provides information on how to install VAX PASCAL on your VAX/VMS operating system.
- The manuals that accompany the VAX/VMS operating system provide full information about the system. The *VAX/VMS Master Index* briefly describes all VAX/VMS system documentation, defines the intended audience for each manual, and provides a synopsis of each manual's contents.

Conventions Used in This Document

This document uses the following conventions.

Convention	Meaning
{ }	Large braces enclose lists from which you must choose one item; for example: $\left\{ \begin{array}{l} \text{expression} \\ \text{statement} \end{array} \right\}$
. . .	A horizontal ellipsis means that the item preceding the ellipsis can be repeated; for example: `digit...`
{ }, . . .	Braces followed by a comma and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with commas; for example: `{label},...`

{ }; . . .	Braces followed by a semicolon and a horizontal ellipsis mean that you can repeat the enclosed item one or more times, separating two or more items with semicolons; for example:
	
	 `REPEAT {statement};...`
	 `UNTIL   expression`

. . .	A vertical ellipsis in a figure or example means that not all of the statements are shown.
[ ]	Square brackets mean that the statement syntax requires the square bracket characters. This notation is used with arrays, sets, and attribute lists; for example: `ARRAY[index1]`

Convention	Meaning
⟦ ⟧	Double brackets enclose items that are optional; for example: `EOLN ⟦(file-variable)⟧`

Items in UPPERCASE letters and special symbols	Uppercase letters and special symbols in syntax descriptions indicate VAX PASCAL reserved words and predeclared identifiers; for example:
	
	`BEGIN`
	`END`

Items in lowercase letters	Lowercase letters represent elements that you must replace according to the description in the text.
#	The number sign denotes special nonprinting characters.
$ PASCAL $_File:	In examples showing commands that you enter and the system's responses, all output lines and prompting characters that the system prints or displays are shown in black leters. All the lines you type are shown in red letters.

In this manual, complex examples and syntax diagrams have been divided into several lines to make them easy to read. PASCAL does not require that you format your programs in any particular way; therefore, you should not regard the formats used in this manual as mandatory.

Developing VAX PASCAL Programs on the VAX/VMS System

Chapter 1

Overview

Pascal is a high-level, structured programming language; it is widely used in the writing of both system and application software. VAX PASCAL offers programmers all the features of standard Pascal, many powerful extensions, and the advantages of the VAX/VMS operating system. VAX PASCAL extensions allow for the use of additional data types, add flexibility when accessing files, permit more ways to compile programs and procedures, including separate compilation of procedures and functions and provides a variety of parameter-handling mechanisms.

1.1 Pascal Standards and Portability Issues

VAX PASCAL contains FIPS-109 (Federal Information Processing Standard) validation support. Currently, there are two Pascal standards:

- American National Standard—ANSI/IEEE770X3.97-1983 (ANSI)
- International Standard—ISO 7185-1983(E) (ISO)

The ISO standard is divided into two levels of standardization, Level 0 and Level 1. The technical differences between the ANSI standard and the ISO standard include the following:

- ANSI does not include conformant arrays. The ISO standard Level 0 does not, but Level 1 does.
- ANSI specifies parameter evaluation order for READ, READLN, WRITE, and WRITELN. The ISO standard does not address this issue.

VAX PASCAL has passed the validation suite for Pascal compilers. It received a CLASS A certificate for both levels of the ISO standard as well as the ANSI standard. CLASS A certificates are given to compilers with a fully conforming implementation.

In addition, VAX PASCAL provides a /STANDARD qualifier that instructs the compiler to generate messages when the compilation unit uses non-standard Pascal features. This feature is an aid to meeting portability goals.

1.2 The Structure of a VAX PASCAL Program

VAX PASCAL programs are structured; that is, they use English-like statements that allow the programmer to make the logical flow efficient and understandable. The structure of a VAX PASCAL program and the wide range of available data structures encourage modular programming.

In modular programming, you can divide the solution to a problem into individual parts that can be developed relatively independently. In standard Pascal, the entire program must be in a single source file.

VAX PASCAL, however, offers the option of writing modules that can be combined with other separately compiled, but logically coordinated, modules for execution as a single program; or writing modules that can be developed independently but used as library modules bound into larger systems at link time.

VAX PASCAL also offers a means for separately compiled modules to communicate with each other by allowing modules to share declarations and definitions. The modules can share variables and routines through the use of global and external identifiers; they can also share variables, routines, constants, and types through the use of environment files.

1.3 Major Features of VAX PASCAL

The three major features of VAX PASCAL include the numerous extensions to standard Pascal, integration into the VAX/VMS Common Language Environment, and the use of the VAX/VMS Architecture Environment.

The numerous extensions to standard Pascal provide for effective general use of the language. These extensions are designed to improve programming efficiency, as well as allow programs to run more efficiently. Some of the extensions include a wider variety of floating-point data types, numerous predeclared routines, enhanced string manipulation capabilities, a separate compilation facility, and more flexibility when accessing files and their records.

VAX PASCAL is integrated into the VAX/VMS Common Language Environment. Included among other capabilities, this environment provides the VAX PASCAL user support for the VAX Procedure Calling Standard, allowing for easy access to all VAX/VMS system routines and external routines written in other VAX languages, access to the VAX/VMS Debugger, full use of the VAX/VMS Linker, and support for three RMS record access methods (sequential, direct and keyed). In addition, VAX PASCAL permits programmers to include data definitions from the VAX Common Data Dictionary.

The VAX PASCAL compiler makes full use of the VAX/VMS Architecture Environment features by using the VAX hardware floating-point and character-string instruction sets and virtual memory capabilities of the VAX/VMS operating system.

The VAX Common Data Dictionary helps eliminate inconsistencies among programs and allows variable declarations to be shared by many programs in multiple programming languages.

Two productivity tools can be purchased separately: the VAX Language-Sensitive Editor (VAXLSE) and the VAX Source Code Analyzer (VAXLSE). VAXLSE provides syntactic models or templates that fit together to form programs that are free of syntax errors. VAXSCA can help you understand the complexities of large-scale software projects by letting you make inquires about the symbols used in such projects. This tool is extremely useful during the implementation and maintenance phases of a development project, regardless of your familiarity with the project.

Additionally, you can tune VAX PASCAL applications using the VAX Performance and Coverage Analyzer to pinpoint execution bottlenecks and other performance problems. This tool provides detailed data in a variety of reporting formats to make performance and test coverage analysis a routine part of every program development cycle.

Chapter 2

Introduction to the VAX/VMS Operating System

When you develop VAX PASCAL programs on a VAX/VMS system, you use commands that are part of the DIGITAL Command Language (DCL). This chapter teaches you what you need to know to use the VAX/VMS system and some common DCL commands. The following topics are discussed:

- Logging in to and out of the VAX/VMS system
- Accessing the HELP facility
- Using DCL commands
- Developing VAX PASCAL programs
- Creating directories and subdirectories
- Using DCL file-handling commands
- Developing command procedures

For a complete list of all of the DCL commands available, see the *VAX/VMS DCL Dictionary.*

2.1 Logging In and Out

Before you can log in to the VAX/VMS operating system, you must have an account set up for you. Your system manager is generally the person responsible for establishing this and will provide you with a user name and a password. Once you are an authorized user, you can regularly access the system.

When you log in to the VAX/VMS operating system, the system prompts you for your user name and password. Both of these are unique to your account and distinguish you from other users on the system. When you enter your user name, each character is displayed, or "echoed", on the screen. However, when you enter your password, no characters are displayed. This enables you to protect your account from others trying to access it. For example:

```
RET
Username: SMITH RET
Password:       RET

$
```

The dollar sign ($) is a symbol that the VAX/VMS system uses as a prompt. When this prompt is displayed, it indicates that the login procedure was successful and that you can begin entering commands. If you incorrectly enter your user name or password, the system displays an error message. To gain access to the system, you must repeat the login procedure.

To change your password, type the SET PASSWORD command and press the RETURN key. The VAX/VMS system prompts you for your current password. Note that when you enter your current password, the VAX/VMS system does not echo it to your terminal. The VAX/VMS system then prompts you for your new password, which can have a maximum of 31 characters composed of the letters A through Z and the numbers 0 through 9, as well as the dollar sign ($) and the underscore (_). Note that when you enter your new password, VAX/VMS does not echo it to your terminal.

To verify that you have entered your password correctly, VAX/VMS prompts you to enter your new password again. If the two new passwords do not match, your original password remains in effect.

```
$ SET PASSWORD          RET
Old password:           RET
New password:           RET
Verification:           RET
$
```

To end the current session, enter the LOGOUT command:

```
$ LOGOUT
```

The system responds with the following message:

```
SMITH logged out at DD-MMM-YYY HH:MM
```

2.2 Accessing the HELP Facility

The VAX/VMS system provides you with an online HELP facility. This can be very useful if you do not have documentation on a particular command readily available. To gain access to the HELP facility, you use the DCL command HELP. For example:

```
$ HELP RET
```

The VAX/VMS system displays a list of topics for which help is available and prompts you for your choice of topic. If you want information on a specific command, such as the PASCAL command, type that command after the topic prompt. For example:

```
Topic? PASCAL RET
```

When this command line is executed, the VAX/VMS system displays a description of the command, lists all VAX PASCAL qualifiers, parameters and other topics for which help is available, and prompts you for a choice of subtopic. For example:

```
PASCAL Subtopic? /DEBUG RET
```

If you know which subtopic you want information on beforehand, you can type the HELP command directly followed by the PASCAL command and the subtopic you want help on. For example:

```
$ HELP PASCAL /DEBUG RET
```

To exit from the HELP facility, press CTRL/Z. Once the dollar sign prompt is displayed, the VAX/VMS system is ready to accept a new command.

2.3 Entering and Editing DCL Commands

Once you have gained access to the system, you can use DCL commands to perform specific tasks. DCL commands are words, generally verbs, that describe the action they perform. Following are the most important rules for entering DCL command lines Other rules are described in the *VAX/VMS DCL Dictionary.*

- You can typically truncate any command name or qualifier name to four characters. Fewer than four characters is acceptable if the truncated name is unique to the command that you want.
- You must precede each qualifier name with a single slash character (/).
- You can enter commands and qualifiers in either uppercase or lowercase letters.
- If you omit a required parameter (for example, a file specification), the DCL command interpreter will prompt you for it.
- You can type a command on as many lines as you wish, as long as you end each line (except the last) with a hyphen (-).
- After you have typed a complete command line, you must press RETURN to execute the command.

If you enter a command incorrectly (for example, if you misspell a command or qualifier name), the command interpreter issues an error message and you must either retype the entire command or edit the command and then reenter it. Command line editing enables you to correct typographical errors and other errors in lengthy command lines and saves you the trouble of retyping the entire line.

To edit DCL command lines, you can use various control characters and keypad keys. The following list describes some of these editing functions:

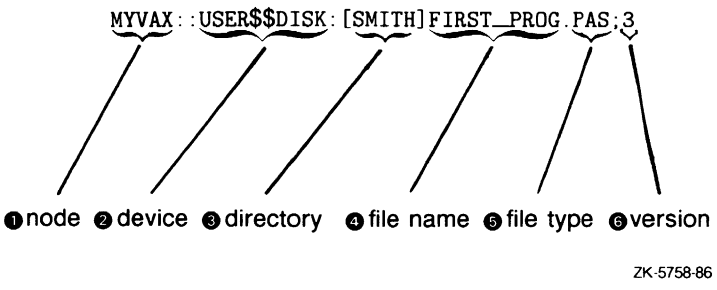
DELETE key	Deletes the last character entered at the terminal.
CTRL/B or up arrow key	Displays the last command line entered. You can display up to twenty previously entered command lines by continuing to press CTRL/B or the up arrow key. To display command lines in the opposite direction, press the down arrow key.
CTRL/D or left arrow key	Moves the cursor one character to the left.
CTRL/F or right arrow key	Moves the cursor one character to the right.
CTRL/U	Deletes characters from the beginning of the line to the cursor.
CTRL/C or CTRL/Y	Cancels or interrupts command processing.
CTRL/A	Switches between overstrike mode and insert mode.

2.4 Understanding the Directory Structure

A directory is a catalog of files. Each file is distinguished by its name, file type, and version number. It is not necessary to specify the complete file specification every time you compile, link, or run a source program. Under most conditions, it is necessary only to specify the file name.

Figure 2–1 illustrates a complete file specification.

Figure 2–1: Complete File Specification


MYVAX::USER$$DISK:[SMITH]FIRST_PROG.PAS;3
❶node ❷device ❸directory ❹file name ❺file type ❻version
ZK-5758-86

❶ Specifies a computer system that is part of a network.

❷ Identifies the hardware device on which the file is either stored or written.

❸ Specifies the directory in which a file is catalogued.

❹ Distinguishes a file from others contained in the same directory. A file name can have up to 39 characters.

❺ Identifies the type of data that a file contains.

❻ Specifies the version number of a file. Each time you edit a file, its version number is incremented by 1.

When you log into the VAX/VMS operating system, by default you are placed in your default directory, which generally has the same name as the account you log into. This default directory is often referred to as your main or top level directory. The VAX/VMS system allows you to create subdirectories from your main directory. Subdirectories are helpful in organizing files. For instance, if you are a multilanguage user, it may be helpful for you to keep all of your VAX PASCAL programs separate from your FORTRAN programs. To create a subdirectory, you enter the DCL command CREATE/DIRECTORY as shown in the following example:

```
$ CREATE/DIRECTORY [SMITH.PASCAL_PROG]
```

To move from one directory to another, you use the SET DEFAULT command:

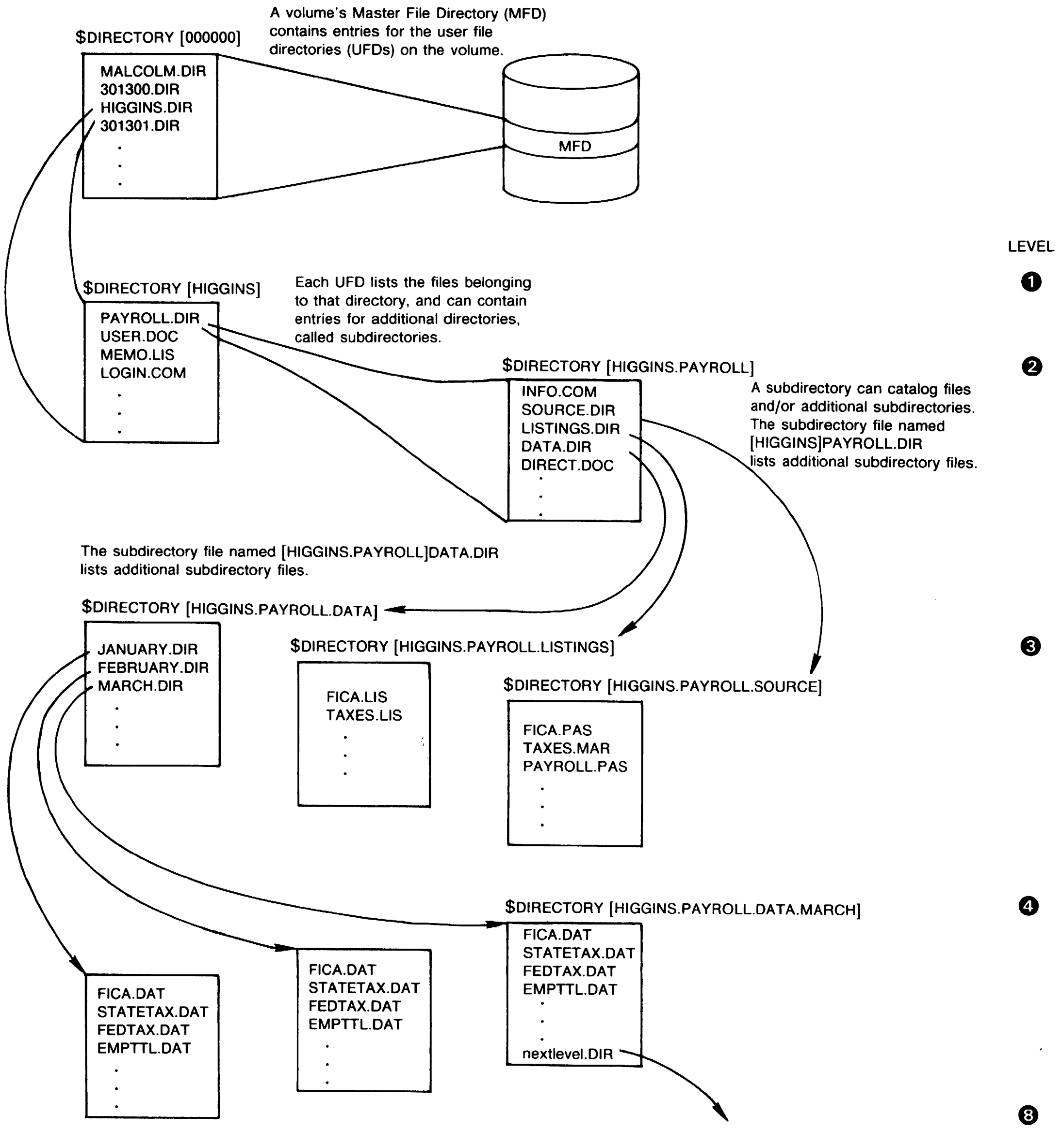
```
$ SET DEFAULT [SMITH.FORTRAN_PROG]
```

The number of directories you are able to create is limited only by the amount of disk space you have available. Figure 2-2 illustrates the concept of directory hierarchies.

To refer to files that are contained in your directory hierarchy but that are not contained in your current default directory, you can use two special symbols: the ellipsis (. . .) and the hyphen, or minus sign (-). The ellipsis is used to search down a directory hierarchy and the hyphen is used to search up a directory hierarchy.

For instance, [SMITH . . .] refers to the directory [SMITH] and all of the subdirectories in the hierarchy below [SMITH]. The directory specification [. . . PASCAL] refers to all subdirectories named PASCAL below the current default directory. Because you can specify the current default directory with empty brackets ([]), you can refer to the entire hierarchy under your current default directory with [. . .].

Figure 2–2: A Directory Hierarchy



$DIRECTORY [000000]
A volume's Master File Directory (MFD) contains entries for the user file directories (UFDs) on the volume.
MALCOLM.DIR
301300.DIR
HIGGINS.DIR
301301.DIR
MFD
LEVEL
1
$DIRECTORY [HIGGINS]
Each UFD lists the files belonging to that directory, and can contain entries for additional directories, called subdirectories.
PAYROLL.DIR
USER.DOC
MEMO.LIS
LOGIN.COM
2
$DIRECTORY [HIGGINS.PAYROLL]
INFO.COM
SOURCE.DIR
LISTINGS.DIR
DATA.DIR
DIRECT.DOC
A subdirectory can catalog files and/or additional subdirectories. The subdirectory file named [HIGGINS]PAYROLL.DIR lists additional subdirectory files.
The subdirectory file named [HIGGINS.PAYROLL]DATA.DIR lists additional subdirectory files.
$DIRECTORY [HIGGINS.PAYROLL.DATA]
JANUARY.DIR
FEBRUARY.DIR
MARCH.DIR
$DIRECTORY [HIGGINS.PAYROLL.LISTINGS]
FICA.LIS
TAXES.LIS
3
$DIRECTORY [HIGGINS.PAYROLL.SOURCE]
FICA.PAS
TAXES.MAR
PAYROLL.PAS
$DIRECTORY [HIGGINS.PAYROLL.DATA.MARCH]
4
FICA.DAT
STATETAX.DAT
FEDTAX.DAT
EMPTTL.DAT
nextlevel.DIR
FICA.DAT
STATETAX.DAT
FEDTAX.DAT
EMPTTL.DAT
FICA.DAT
STATETAX.DAT
FEDTAX.DAT
EMPTTL.DAT
8

ZK-5781-HC

Hyphens allow you to search up the hierarchy one directory at a time; each hyphen stands for one level. If, for example, your current default directory is [SMITH.PASCAL_PROG], you can refer to [SMITH] by specifying [-].

Every subdirectory is maintained by a file in the directory above it that has a DIR file type. To delete a subdirectory, you must first remove any files that are contained in it. You can then delete the DIR file for that subdirectory.

Note that you can use square brackets ([]) and angle brackets (<>) interchangeably to refer to directories.

For a more detailed discussion of the directory structure, see the *VAX/VMS DCL Concepts Manual.*

2.5 Using DCL File-Handling Commands

DCL file-handling commands allow you to modify and maintain your source programs. The following sections describe how to use DCL commands to perform basic file operations such as displaying files, printing and typing files, deleting files, purging files, renaming and moving files, searching files, and setting file protection.

For more information on the commands discussed in this section and their qualifiers, see the *VAX/VMS DCL Dictionary* or type HELP followed by the command name at the DCL prompt.

2.5.1 Displaying Files

To display a list of all of the files that are contained in a directory, you use the DIRECTORY command. When you enter the DIRECTORY command with no parameters or qualifiers, the VAX/VMS system displays an entire list of all of the files that are contained in your current directory. For example:

```
$ DIRECTORY

Directory USER$$DISK:[SMITH]

PROG1.DIA;22      PROG1.EXE;2        PROG1.OBJ;3        PROG1.OBJ;5
PROG1.OBJ;53      PROG1.PAS;53
```

If you want to know how many versions of a specific file exist in your current directory, you enter the DIRECTORY command along with a specific file name and file type as shown in the following example:

```
$ DIRECTORY PROG1.PAS

Directory USER$$DISK:[SMITH]

PROG1.PAS;53

Total of 1 file.
```

The VAX/VMS system displays a list of all the versions of the file that currently exist.

However, if you want to view only a selected group of files that contain a specific file type, you can use an asterisk (*) and percent sign (%) individually or together as wildcard characters. An asterisk signifies any number of characters including zero, whereas a percent sign signifies exactly one character. Therefore, *.PAS represents all files that end with the file type, PAS, whereas 3%.PAS represents only those files that contain two characters for a file name, the first of which is a 3, and have the .PAS file type. The following example uses an asterisk as a wildcard character.

```
$ DIRECTORY  *.OBJ

Directory USER$$DISK:[SMITH]

PROG1.OBJ;3        PROG1.OBJ;5        PROG1.OBJ;53

Total of 3 files.
```

The VAX/VMS system displays only those files that end with the file type OBJ.

2.5.2 Printing and Typing Files

To examine the contents of a file, you can use either the PRINT command or the DCL TYPE command. The PRINT command outputs the contents of a file to a printer, and the TYPE command outputs the contents of a file to your screen.

In the following example, the PRINT command outputs the file PROG1.PAS to a printer:

```
$ PRINT PROG1.PAS
```

By default, the VAX/VMS system prints the most recently created version of PROG1.PAS. To print a specific version of PROG1.PAS, you must specify a version number. For instance, in the following example, the VAX/VMS system prints the third version of PROG1.PAS.

```
$ PRINT PROG1.PAS;3
```

With the PRINT command, you can specify command qualifiers. For example, the /AFTER qualifier allows you to specify that the job not be printed until a specific time of day.

With the TYPE command, you can display the contents of a file on your screen. If you specify the /PAGE qualifier, you can display one screen of text at a time. If you do not specify the /PAGE qualifier, you can press CTRL/S to interrupt the display for careful examination of a particular section and then resume the display by pressing CTRL/Q. If your keyboard has a NOSCROLL or HOLD SCREEN key, you can use it to toggle between interrupting and resuming the display.

2.5.3 Deleting Files

To delete a file, you use the DELETE command. With the DELETE command, you must specify the file name, the file type, and the version number of the file you want to delete. For instance, in the following example, the VAX/VMS system deletes the first version of PROG1.PAS.

```
$ DELETE PROG1.PAS;1
```

If you want to delete all versions of a file, you can use an asterisk (*) as a wildcard character. For example:

```
$ DELETE PROG1.PAS;*
```

Similarly, if you want to delete all files with a particular file type, you can use asterisks (*) and percent signs (%) as wildcard characters. For instance, in the following example, the VAX/VMS system deletes all files that have the file type LIS:

```
$ DELETE *.LIS;*
```

You can specify command qualifiers with the DELETE command. For instance, if you specify /CONFIRM, a request is issued before each individual DELETE operation so that you can confirm that the operation should be performed on a particular file.

2.5.4 Purging Files

The PURGE command deletes all versions of files except the most recently created versions. Unlike the DELETE command, you can specify the PURGE command without specifying a file name or file type; the VAX/VMS system deletes all files except the most recent version in your current directory.

```
$ PURGE
```

You can specify the number of versions that are actually deleted by specifying the /KEEP qualifier. This command qualifier specifies the maximum number of versions of a specified file to be kept in your directory. For instance, in the following example, VAX/VMS deletes all but the two highest versions of PROG1.PAS.

```
$ PURGE PROG1.PAS/KEEP=2
```

2.5.5 Renaming and Moving Files

To change the identification of one or more files, use the RENAME command. For instance, in the following example, the most recently created version of PROG1.PAS is renamed SECOND_PROG.PAS.

```
$ RENAME PROG1.PAS SECOND_PROG.PAS
```

You can also use the RENAME command to move a file from one directory to another. For instance, in the following example, the file SECOND_PROG. PAS is moved from the directory [SMITH] to the subdirectory [SMITH.PASCAL_PROG]:

```
$ RENAME [SMITH]SECOND_PROG.PAS [SMITH.PASCAL_PROG]
```

To move an entire set of files that have a common file name or file type, you can use the asterisk (*) as a wildcard character. For instance, when executed, the following command line changes the directory name of all files that contain the file name SECOND_PROG.

```
$ RENAME [SMITH]SECOND_PROG.*;* [SMITH.PASCAL_PROG]*.*;*
```

2.5.6 Searching Files

To search a file or a group of files for a specified string, use the SEARCH command. This command allows you to search the contents of a specified file or group of files for a particular string or strings and lists all of the lines in which the string or strings appear. The following command causes the entire directory [SMITH.PASCAL_PROG] to be searched for any files of type PAS that contain the string "Course_Data":

```
$ SEARCH [SMITH.PASCAL_PROG]*.PAS;* "Course_Data"
```

2.5.7 Setting File Protection

To prevent others from gaining unwanted access to a file, you use the SET PROTECTION command. With this command, you specify user categories with access types. Table 2–1 lists the four user categories and Table 2–2 lists the four file access types.

Table 2–1: User Categories

User Category	Description
OWNER	The user who created the file
GROUP	All users, including the owner, who have the same group number in their user identification codes (UIC) as the owner of the file
WORLD	All users
SYSTEM	All users who have the same system privilege (SYSPRV) or low group numbers (from 1 through 10)

Table 2–2: File Access Types

Access	Description
READ	The ability to examine, print, or copy a file
WRITE	The ability to modify or write a file
EXECUTE	The ability to execute a file that contains executable program images
DELETE	The ability to delete a file

When you specify the SET PROTECTION command, the class names and access types can be spelled out or abbreviated to the first letter. For example, you may want to give system users and your project members complete access to the file PROG1.PAS, and give others read access only. To do this, you would enter the following command:

```
$ SET FILE PROG1.PAS/PROT=(S:RWED,O:RWED,G:RWED,W:R)
```

2.6 Using Command Procedures

Command procedures are composed of DCL commands that are grouped as one unit. Each command line is executed by the DCL command interpreter. You can use command procedures to generate sequences of command lines that you type frequently. For example, you could create a command procedure that compiles, links, and runs your source programs while checking for error status, or you could create a command procedure that deletes all files that end with a particular file type. Instead of entering each command in the group separately, you would simply execute the command procedure.

The following sections briefly discuss the rules for defining DCL symbols and logical names for use in command procedures, as well as how to create and execute a command procedure. In addition, a sample command procedure and a sample LOGIN.COM file are provided. For a more detailed description, see the *Guide to Using DCL and Command Procedures on VAX/VMS.*

2.6.1 Defining DCL Symbols and Logical Names

You define a symbol by placing the equal sign (=) or double equal sign (==) assignment operator between the symbol and the string it represents. If you use the equal sign operator, the VAX/VMS system inserts the symbol in a local symbol table; if you use the double equal sign operator, the VAX/VMS system inserts the symbol in the global symbol table. A local symbol exists as long as the command level at which it is defined remains active, unless the symbol is specifically deleted. A global symbol exists for the duration of the process, unless it is specifically deleted or treated as undefined.

For example, the following command defines the global symbol PASCAL as a command to invoke the VAX PASCAL compiler, with two added qualifiers:

```
$ PASCAL == "PASCAL/DEBUG/LIST"
```

The *VAX/VMS DCL Dictionary* describes symbol tables in detail. Briefly, the VAX/VMS system creates a global symbol table for you when you log in. Then, each time you execute a command procedure, the VAX/VMS system creates a new command level and a local symbol table for that level. If one command procedure executes another procedure, the VAX/VMS system creates another new—lower—command level and another local symbol table, and so forth. Every command level can access the symbols in the global symbol table.

Note that the VAX/VMS system discards local symbols and their values when a procedure exits. That is, a program or command procedure can create local symbols, but the symbols disappear when the program or procedure ends. However, you can define local symbols at DCL command level, and these symbols remain until you explicitly remove them or until you log out.

You can pass information to a higher-level command procedure by defining a global symbol to contain the information. Because there is only one global symbol table, and it is accessible at all command levels, the higher-level command procedure can simply test the value of the symbol.

To allow abbreviations of a symbol, use the asterisk (*) character to end the acceptable abbreviation. For example, if you wanted to abbreviate the command PASCAL to PAS, you could use the following command:

```
$ PAS*CAL == "PASCAL/DEBUG/LIST"
```

Once you have defined this symbol, you can type PAS, plus any remaining letters in the command if you want, to invoke the VAX PASCAL compiler with the /DEBUG and /LIST qualifiers. Note that the double equal sign in this symbol definition causes the creation of a global symbol.

You can determine the definition of a symbol by typing SHOW SYMBOL followed by the symbol name. For example:

```
$ SHOW SYMBOL PAS
  PAS*CAL = PASCAL/DEBUG/LIST
```

To delete a symbol, type DELETE/SYMBOL, followed by the symbol name. If you do not specify a symbol table qualifier, /LOCAL is assumed. If the symbol is in the global table, type DELETE/SYMBOL/GLOBAL followed by the symbol name. For example, you could delete the symbol PASCAL that you had previously defined, by typing the following command:

```
$ DELETE/SYMBOL/GLOBAL PASCAL
```

To create a logical name, you use either the DCL command DEFINE or the DCL command ASSIGN. A logical name is a name that is equated to an equivalence string, or a list of equivalence strings. For example, the following command assigns the logical name INFO to the file specification USER$$DISK:[SMITH]INFO.DAT;12:

```
$ DEFINE INFO USER$$DISK:[SMITH]INFO.DAT;12
```

You can determine the definition of a logical name by typing SHOW LOGICAL, followed by the logical name. For example:

```
$ SHOW LOGICAL INFO
  "INFO" = "USER$$DISK:[SMITH]INFO.DAT;12" (LNM$PROCESS_TABLE)
```

When you create a logical name, it is maintained in a logical name table. For more information on logical name tables, see the *VAX/VMS DCL Dictionary*.

You can keep a file of symbols and logical names for the system to define every time you log in, thus creating a set of personal commands for special purposes. For information on login command procedures, see Section 2.6.4.

2.6.2 Creating and Executing Command Procedures

To create a command procedure, you can invoke a text editor or you can use the DCL command CREATE. If you use the default file type COM, you need not include the file type when you execute the procedure.

Once you have created your command procedure, you can execute it either interactively or by batch. To execute a command procedure interactively, you must specify the name of the file preceded by the Execute Procedure (@) command. For instance, in the following example, the procedure SAMPLE.COM in the current directory is executed.

```
$ @SAMPLE
```

To execute a command procedure by batch, you use the DCL command SUBMIT. This command allows you to submit your procedure to a system batch job queue for execution. Once the job completes, the system prints a log file indicating the status of the job and then deletes the file from your directory.

In the following example, SAMPLE.COM is placed in the system batch job queue. A log file named SAMPLE.LOG will be created.

```
$ SUBMIT SAMPLE
```

2.6.3 Sample Command Procedure

The following command procedure shows how command procedures can aid in the programming process. This command procedure incorporates many of the DCL commands discussed throughout this chapter. Any text that follows an exclamation point (!) is interpreted by the DCL command interpreter as a comment.

```
$ !The following command procedure tests to see if the file name you
$ !input exists.  If it does exist, the procedure will automatically
$ !compile, link, and run your source program.  If the file does not
$ !exist, the procedure will ask you if you want to create the file.
$ !If you respond "yes", the supplied source file is created and the
$ !default text editor is invoked.
$ !
$ !
$ IF P1 .EQS. "" THEN INQUIRE P1 "FILE NAME"
$ FILETYPE = ".PAS"
$ FILESPEC = P1 + FILETYPE                                            ❶
$ IF F$SEARCH(FILESPEC) .EQS. "" THEN GOTO MESSAGE
$ SHOW SYMBOL FILESPEC
$ !
$ !
$ !
$ ON ERROR THEN GOTO PRINT_ROUTINE
$ WRITE SYS$OUTPUT "Compiling..."                                     ❷
$ PASCAL/LIST 'P1'
$ !
$ !
$ WRITE SYS$OUTPUT "Linking..."                                       ❸
$ LINK 'P1'
$ !
$ !
$ WRITE SYS$OUTPUT "Running..."                                       ❹
$ DEFINE/USER_MODE SYS$INPUT SYS$COMMAND
$ RUN 'P1'
$ GOTO END
$ SET NOON
$ !
$ !
$ PRINT_ROUTINE:                                                    ❺
$   WRITE SYS$OUTPUT "Error in program - processing stops"
$   PRINT 'P1'.lis
$ END:
$ !
$ PURGE
$ !
$ EXIT

$ !
$ MESSAGE:                                                          ❻
$   WRITE SYS$OUTPUT "Can't find requested file"
$   INQUIRE ANS "Do you want the file created?"
$   IF ANS .EQS. "YES" THEN GOTO EDIT_ROUTINE
$ EXIT
$ !
$ EDIT_ROUTINE:                                                     ❼
$   DEFINE/USER_MODE SYS$INPUT SYS$COMMAND
$   WRITE SYS$OUTPUT "Invoking editor..."
$   EDIT 'P1'.PAS
$ EXIT
```

❶ Labels the beginning of the command procedure and prompts the user for a source file name if one has not been supplied. If the file is not found, control is transferred to a routine labeled MESSAGE.

❷ Marks the start of the compilation process. The procedure outputs an informational message and invokes the VAX PASCAL compiler.

❸ Marks the start of the linking process. The procedure outputs an informational message and invokes the linker.

❹ Labels the start of the running process. The procedure outputs an informational message and runs the program. The input device is equated to the user's terminal to allow data to be entered.

❺ Identifies the actions of the command procedure in the event an error occurs during execution. When an error does occur, control is transferred to this routine. The procedure outputs an informational message and prints the listing file.

❻ Labels the routine that alerts the user that the source file has not been found. If the user chooses to create the file, control is transferred to the routine labeled EDIT_ROUTINE.

❼ Marks the start of the edit routine. Again, the input device is equated to the user's terminal to allow data to be entered. The procedure then outputs an informational message and creates the supplied source file by invoking the user's default text editor.

2.6.4 Login Command Procedures

If you are a frequent user of the VAX/VMS system, you may find that you are entering the same long command lines on a regular basis. To avoid such repetition, you can create a special command procedure that equates these commands and statements to symbols and logical names. This special command procedure is called a LOGIN.COM file.

When you log in, the VAX/VMS system automatically searches your default device and directory for a LOGIN.COM file. If the file exists, the system automatically executes the commands within that file.

A typical LOGIN.COM file might appear as follows:

```
$ !If the current process is noninteractive then exit
$ IF F$MODE().NES. "INTERACTIVE" THEN EXIT
$ !
$ !*******************************************
$ !*       Convenient Commands               *
$ !*******************************************
$ !
$ PURGE == "PURGE/KEEP=2/CONFIRM"
$ ST == "SHOW TIME"
$ QUE == "SHOW QUEUE"
$ M*AIL == "MAIL"
$ !
$ !*******************************************
$ !*      Symbols for my directories         *
$ !*******************************************
$ !
$ HOME == "SET DEFAULT [SMITH]"
$ WORK == "SET DEFAULT [SMITH.WORKSP]"
$ EXAMPLES == "SET DEFAULT [SMITH.PROG_EXAMPLES]"
$ !
$ !********************************************************
$ !* Logical names for the people I frequently send mail to *
$ !********************************************************
$ !
$ DEFINE JEN MYVAX::BROWN
$ DEFINE SUE MYVAX::KELLY
$ DEFINE BOB MYVAX::CONNELL
$ EXIT
```

For more information on creating a login command file, see the *Guide to Using DCL and Command Procedures on VAX/VMS.*

Chapter 3

VAX PASCAL Program Development

This chapter describes how to create, compile, link, and run a VAX PASCAL program using DCL commands.

3.1 Creating a VAX PASCAL Program

To create and modify a VAX PASCAL program, you must invoke a text editor. VAX/VMS provides you with two text editors: VAX EDT and the VAX Text Processing Utility (VAXTPU). The following sections describe briefly how to use VAX EDT and VAXTPU.

3.1.1 Using VAX EDT

VAX EDT is an interactive general-purpose text editor that offers three editing modes: keypad, nokeypad, and line. Keypad mode uses the numeric keypad that appears to the right of your main keyboard. With nokeypad mode, you type commands on a command line, which EDT processes when you press RETURN. Line mode focuses on the line as the unit of text. The appearance of a line mode asterisk prompt indicates that you can type a line mode command. When you begin your editing session, editing in line mode is the default. Unlike line mode, keypad mode and nokeypad mode continually display the contents of the file on your screen.

The following command line, when executed, invokes the EDT editor and creates the file PROG_1.PAS.

```
$ EDIT/EDT PROG_1.PAS
```

To change from line mode to keypad mode, enter the CHANGE command at the asterisk prompt. To return to line mode from keypad mode, press CTRL/Z. To change from line mode to nokeypad mode, enter the SET NOKEYPAD command and then the CHANGE command.

If you are in the middle of an editing session and your system fails, you can recover your edits by reentering the EDIT command followed by the /RECOVER qualifier. EDT recreates your last editing session on your screen up to the point where it was interrupted using the contents of a journal file that is maintained during the editing session.

VAX EDT offers an online HELP facility that you can access during an editing session. In line mode, you can enter the HELP command. VAX EDT displays general information on EDT as well as detailed information on both line mode editing and nokeypad mode editing. When you are in keypad mode, you can press the HELP key or the PF2 key; VAX EDT displays a keypad diagram on your screen and a list of keypad editing keys. For help on a specific keypad function, press the key you want help on.

For more detailed information on how to use the VAX EDT editor, see the *VAX EDT Reference Manual.*

3.1.2 Using VAXTPU

The VAX Text Processing Utility (VAXTPU) is a high-performance, programmable utility. Like VAX EDT, VAXTPU provides you with an online HELP facility that you can access during your editing session. It also provides you with a journal facility. Unlike VAX EDT, VAXTPU provides you with multiple windows; this feature allows you to view two files on your screen at the same time.

With VAXTPU, you can use editing interfaces to edit your VAX PASCAL programs. VAXTPU provides you with two editing interfaces: the Extensible VAX Editor (EVE) and the VAXTPU EDT Keypad Emulator. You can also create your own interfaces. The following sections describe briefly how to use the EVE interface and the EDT Keypad Emulator interface.

3.1.2.1 The EVE Interface

The Extensible VAX Editor (EVE) is an interactive text editor that lets you execute common editing functions by using the EVE keypad, or execute more advanced functions by entering commands on the EVE command line. The following command line, when executed, invokes the EVE editor and creates the file PROG_1.PAS.

```
$ EDIT/TPU PROG_1.PAS
```

You can define a global symbol for the EDIT/TPU command by placing a symbol definition in your LOGIN.COM file:

```
$ EVE == "EDIT/TPU"
```

Once this command line is executed, you can type EVE at the DCL prompt followed by the name of the file you want to modify or create.

For more information on using the advanced features of EVE, see the *Guide to Text Processing on VAX/VMS.*

3.1.2.2 The EDT Keypad Emulator Interface

The EDT Keypad Emulator interface provides all of the functions associated with VAX EDT and uses the same keys to perform each function. To access the EDT Keypad Emulator, you enter the following command line:

```
$ EDIT/TPU/SECTION=EDTSECINI.GBL
```

You can define a global symbol for this command by placing a symbol definition in your LOGIN.COM file:

```
$ EDTEM == "EDIT/TPU/SECTION=EDTSECINI"
```

When this command line is executed, you can type EDTEM at the DCL prompt followed by the name of the file you want to create or modify. For example:

```
$ EDTEM PROG_1.PAS
```

For more detailed information on how to use the VAX EDT Keypad Emulator, see the *VAXTPU EDT Keypad Emulator Quick Reference Guide.*

3.2 Compiling a VAX PASCAL Program

The primary functions of the VAX PASCAL compiler are to verify VAX PASCAL source statements, to issue error messages, and to generate and group machine language instructions into an object module for the linker.

3.2.1 PASCAL Command

To compile a source program or module, use the PASCAL command. It has the following form:

```
PASCAL [[{/command-qualifier}...]]{file-spec[[{/file-qualifier}...]]}{+ ,}...
```

/command-qualifier
The name of a qualifier (see Section 3.2.2) that indicates special processing to be performed by the compiler on all files listed.

file-spec
The name of an input source file that contains the program or module to be compiled. If you separate multiple source file specifications with commas, the programs are compiled separately. If you separate the file specifications with plus signs, the files are concatenated and compiled as one program.

/file-qualifier
The name of a qualifier (see Section 3.2.2) that indicates special processing to be performed by the compiler on the files to which the qualifier is attached.

In interactive mode, you can issue the PASCAL command without an accompanying specification for the input file. The system responds with a prompt for the file specification:

```
$ PASCAL RET
_File:
```

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

3.2.2 PASCAL Command Qualifiers

In many cases, the PASCAL command's defaults are sufficient for compilation. However, when you need special processing, you can specify PASCAL qualifiers to override the defaults.

Compiler qualifiers can apply either to the PASCAL command or to the specification of the file being compiled. When a qualifier follows the PASCAL command, it applies to all the files listed.

A PASCAL qualifier is often followed by additional information that pertains to the qualifier's action. A PASCAL qualifier has the following form:

```
                       ( file-spec              )
                       )  identifier            (
/qualifier  [[   =    <  (identifier, . . . )    >      ]]
                       (  n                     )
```

qualifier
The name of a PASCAL qualifier, as listed in Table 3–1.

file-spec
The name of an output file; used with the /ENVIRONMENT, /LIST /OBJECT, /DIAGNOSTICS, and /ANALYSIS_DATA qualifiers only.

identifier
The name of one or more options that modify the /CHECK, /DEBUG, /OPTIMIZE, /SHOW, /STANDARD, /TERMINAL, and /USAGE qualifiers only.

n
An integer constant that indicates the maximum number of errors to be detected before compilation ceases; used with the /ERROR_LIMIT qualifier only.

You should use full qualifier names in command procedure files to ensure readability. You can, however, 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. To guarantee compatibility with future releases of the system, you should not use fewer than four characters. For example, you could abbreviate /CROSS_REFERENCE to /CROS and /NOWARNINGS to /NOWARN.

Example 1

```
$ PASCAL CALC
```

The source file CALC.PAS is compiled using the default qualifiers.

Example 2

```
$ PASCAL/CHECK/STANDARD CALC
```

The source file CALC.PAS is compiled, and checking code is generated for all checking options. In addition, the compiler issues warnings for the use of language extensions.

Example 3

```
$ PASCAL/LIST/CROS CALC
```

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

Example 4

```
$ PASCAL/LIST A, B, C+D
```

The /LIST qualifier in this example causes three listing files to be produced, namely A.LIS, B.LIS, and C.LIS (which contains the listings for both C and D).

Example 5

When a qualifier follows the file specification, it applies only to the file immediately preceding it. Files concatenated with plus signs are considered one file. This command line produces one listing file, D.LIS:

```
$ PASCAL A, B, C+D/LIST, F
```

Example 6

By using command and file qualifiers on the same command line, you can override the qualifier's effect for selected files. In this example, the PASCAL command produces two listing files, A.LIS and C.LIS:

```
$ PASCAL/LIST A, B/NOLIST, C+D
```

The /NOLIST qualifier overrides the effect of /LIST on the PASCAL command; no listing file for B is produced.

Table 3–1 lists the qualifiers and defaults that apply to the PASCAL command.

Table 3–1: PASCAL Command Qualifiers

Qualifier	Action	Default
/ANALYSIS_DATA[[=file-spec]] /NOANALYSIS_DATA	Creates a file containing source code analysis information; /NOANALYSIS_DATA suppresses the production of an analysis file.	/NOANALYSIS_DATA
/CHECK [[=({option-list},...)]] /NOCHECK	Generates code to perform run-time checks; /NOCHECK prevents run-time checking.	/CHECK=BOUNDS
/CROSS_REFERENCE /NOCROSS_REFERENCE	Produces a cross-reference listing; /NOCROSS_REFERENCE suppresses the cross-reference listing.	/NOCROSS_REFERENCE
/DEBUG [[=({option-list},...)]] /NODEBUG	Generates data for the VAX/VMS Debugger; /NODEBUG suppresses debugging data.	/DEBUG=TRACEBACK
/DIAGNOSTICS [[=file-spec]] /NODIAGNOSTICS	Creates a file containing compiler messages and diagnostic information; /NODIAGNOSTICS suppresses the production of a diagnostics file.	/NODIAGNOSTICS
/ENVIRONMENT [[=file-spec]] /NOENVIRONMENT	Produces an environment file; /NOENVIRONMENT suppresses the production of an environment file.	Determined by ENVIRONMENT attribute in source program
/ERROR_LIMIT [[=N]] /NOERROR_LIMIT	Terminates compilation after the specified number of errors; /NOERROR_LIMIT continues compilation until 500 errors have been detected.	/ERROR_LIMIT=30
/G_FLOATING /NOG_FLOATING	Specifies G_floating representation and machine instructions for values of type DOUBLE; /NOG_FLOATING specifies D_floating instructions.	/NOG_FLOATING

Table 3–1 (Cont.): PASCAL Command Qualifiers

Qualifier	Action	Default
file-spec/LIBRARY	Identifies a text library file.	None
/LIST ⟦=file-spec⟧ /NOLIST	Produces a source listing file; /NOLIST suppresses the source listing.	/NOLIST (interactive) /LIST=input-file.LIS (batch)
/MACHINE_CODE /NOMACHINE_CODE	Includes representation of machine code in the source listing file; /NOMACHINE_CODE suppresses the inclusion of machine code.	/NOMACHINE_CODE
/OBJECT ⟦=file-spec⟧ /NOOBJECT	Specifies the name of the object file; /NOOBJECT suppresses the object file.	/OBJECT=input-file.OBJ
/OLD_VERSION /NOOLD_VERSION	Specifies compilation conforming to the VAX PASCAL Version 1 language definition; /NOOLD_VERSION cancels compilation under Version 1 rules.	/NOOLD_VERSION
/OPTIMIZE ⟦=({option-list},...)⟧ /NOOPTIMIZE	Requests the compiler to generate more efficient code by optimizing the compiled program; /NOOPTIMIZE prevents optimization.	/OPTIMIZE
/SHOW ⟦=({option-list},...)⟧ /NOSHOW	Specifies the list of items to be included in the listing file; /NOSHOW suppresses a listing file.	/SHOW=(DICTIONARY, HEADER, INCLUDE, SOURCE, STATISTICS, TABLE_OF_CONTENTS)

Table 3–1 (Cont.): PASCAL Command Qualifiers

Qualifier	Action	Default
/STANDARD ⟦=({option-list},...)⟧ /NOSTANDARD	Prints messages identifying use of VAX PASCAL extensions; /NOSTANDARD suppresses these messages.	/NOSTANDARD
/TERMINAL ⟦=({option-list},...)⟧ /NOTERMINAL	Specifies a list of items to be displayed on the terminal; /NOTERMINAL prevents a list from being displayed.	/NOTERMINAL
/USAGE ⟦=({option-list},...)⟧ /NOUSAGE	Directs the compiler to perform compile-time checks; /NOUSAGE prevents the compiler from performing compile-time checks.	/USAGE=UNINITIALIZED
/WARNINGS /NOWARNINGS	Prints diagnostics for warning level errors; /NOWARNINGS suppresses these messages.	/WARNINGS

3.2.2.1 /ANALYSIS_DATA Qualifier

The /ANALYSIS_DATA qualifier creates a file containing source code analysis information. If you omit the file specification, the analysis file defaults to the name of your source file with a file type of ANA.

The source code analysis file is reserved for use with DIGITAL layered products such as, but not limited to, the VAX Source Code Analyzer.

3.2.2.2 /CHECK Qualifier

The /CHECK qualifier directs the compiler to generate code to perform run-time checks. A single identifier or a list of identifiers enclosed in parentheses may follow /CHECK; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to check.

The system issues an error message and normally terminates execution if any of the conditions in the option list occur. Table 3–2 lists the available checking options, their corresponding actions, and their negations. A negated option suppresses a checking feature.

Table 3–2: /CHECK Qualifier Options

Option	Action	Negation
ALL	Generates checking code for all options	NONE
BOUNDS	Verifies that an index expression is within the bounds of an array's index type and that character-string sizes are compatible with the operations being performed	NOBOUNDS
CASE_SELECTORS	Verifies that the value of a case selector is contained in the corresponding case-label list	NOCASE_SELECTORS
OVERFLOW	Verifies that the result of an integer computation does not exceed the machine representation	NOOVERFLOW
POINTERS	Verifies that the value of a pointer variable is not NIL	NOPOINTERS
SUBRANGE	Verifies that values assigned to variables of subrange types are within the subrange; verifies that a set expression is assignment compatible with a set variable	NOSUBRANGE

BOUNDS is the only checking option enabled by default. The /CHECK qualifier without options is equivalent to /CHECK=ALL. The negation /NOCHECK is equivalent to /CHECK=NONE.

The CHECK attribute in the source program or module overrides the /CHECK qualifier in the command line (see Section 3.2.5.1).

3.2.2.3 /CROSS_REFERENCE Qualifier

The /CROSS_REFERENCE qualifier produces a cross-reference listing of all identifiers. The compiler generates separate cross-references for each routine. Note that this qualifier is ignored if the /LIST qualifier is disabled.

By default, /NOCROSS_REFERENCE is enabled.

3.2.2.4 /DEBUG Qualifier

The /DEBUG qualifier specifies that the compiler is to generate information for use by the VAX/VMS Debugger and the run-time error traceback mechanism. A single identifier or a list of identifiers enclosed in parentheses may follow /DEBUG; these identifiers are the names of options that inform the compiler which type of information it should generate.

Table 3–3 lists the available options, their corresponding actions, and their negations. A negated option suppresses a debugging feature.

Table 3–3: /DEBUG Qualifier Options

Option	Action	Negation
ALL	Specifies that the compiler should include symbol and traceback information in the object module	NONE
SYMBOLS	Specifies that the compiler should include in the object module symbol definitions for all identifiers in the compilation	NOSYMBOLS
TRACEBACK	Specifies that the compiler should include in the object module traceback information permitting virtual addresses to be translated into source program routine names and compiler-generated line numbers	NOTRACEBACK

TRACEBACK is the only debugging option enabled by default. When you specify SYMBOLS without TRACEBACK, the table of compiler-generated line numbers is omitted from the debugger symbol table.

The /DEBUG qualifier without options is equivalent to /DEBUG=ALL. The negation /NODEBUG is equivalent to /DEBUG=NONE. See Chapter 4 for more information on debugging.

3.2.2.5 /DIAGNOSTICS Qualifier

The /DIAGNOSTICS qualifier creates a file containing compiler messages and diagnostic information. If you omit the file specification, the diagnostics file defaults to the name of your source file with a file type of DIA.

The diagnostics file is reserved for use with DIGITAL layered products such as, but not limited to, the VAX Language-Sensitive Editor.

3.2.2.6 /ENVIRONMENT Qualifier

The /ENVIRONMENT qualifier produces an environment file in which declarations and definitions made at the outermost level of a compilation unit are saved. The default file name is the same as the source file name. The default file type is PEN, an abbreviation for "PASCAL Environment." You can provide a different name for the environment file by including a VAX/VMS file specification after the /ENVIRONMENT qualifier, as in /ENVIRONMENT=MASTER.PEN. The use of environment files is described in the *VAX PASCAL Reference Manual*.

The /ENVIRONMENT qualifier in the command line overrides the ENVIRONMENT attribute in the source program or module (see Section 3.2.5.2). Thus, you can suppress an already created environment file by using /NOENVIRONMENT at compile time. Likewise, you can use the /ENVIRONMENT qualifier to change the name of an environment file by providing a file specification that differs from the one provided with the ENVIRONMENT attribute.

By default, the attributes of the source program or module determine whether an environment file is created; however, if the /ENVIRONMENT qualifier is specified at compile time, an environment file will always be created.

3.2.2.7 /ERROR_LIMIT Qualifier

The /ERROR_LIMIT qualifier terminates compilation after the occurrence of a specified number of error messages, excluding warning-level and information-level errors.

If you specify /NOERROR_LIMIT, compilation continues until 500 errors have been detected.

By default, /ERROR_LIMIT=30 is enabled.

3.2.2.8 /G_FLOATING Qualifier

The /G_FLOATING qualifier directs the compiler to use the G_floating representation and instructions for values of type DOUBLE. The negation of this qualifier specifies the use of the D_floating representation and instructions.

If the use of the /G_FLOATING qualifier conflicts with a double-precision attribute specified in the source program or module (see Section 3.2.5.3), a warning occurs.

Routines and compilation units between which double-precision quantities are passed should not mix the D_floating and G_floating data types. Not all VAX processors support the G_floating data type.

By default, /NOG_FLOATING is enabled.

3.2.2.9 file-spec/LIBRARY Qualifier

The /LIBRARY qualifier specifies that a file is a text library file. The text library file specification is required. The text library files in a list of source files must be concatenated by plus signs. The default type is TLB. For more information see the *VAX PASCAL Reference Manual.*

3.2.2.10 /LIST Qualifier

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

The compiler does not produce a listing file in interactive mode unless you specify the /LIST qualifier. In batch mode, the compiler produces a listing file by default. To suppress the listing file, use the /NOLIST qualifier.

In either mode, the listing file is not automatically printed. You must use the PRINT command to obtain a line printer copy of the listing file. A sample listing is shown in Section 3.2.8.

3.2.2.11 /MACHINE_CODE Qualifier

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. The VAX PASCAL compiler does not generate object code if it detects errors in the source program or module.

By default, /NOMACHINE_CODE is enabled.

3.2.2.12 /OBJECT Qualifier

The /OBJECT qualifier specifies the name of the object file. If you omit the file specification, the object file defaults to the name of the first source file listed and has a file type of OBJ.

During the early stages of development, you may find it helpful to suppress the production of object files with the /NOOBJECT qualifier until the source program or module compiles without errors.

By default, /OBJECT is enabled.

3.2.2.13 /OLD_VERSION Qualifier

The /OLD_VERSION qualifier directs the compiler to resolve differences between VAX PASCAL Version 1 and subsequent versions by using the Version 1 definition.

The use of this qualifier causes several changes in the language definition. See Appendix D for descriptions of these changes.

By default, /NOOLD_VERSION is enabled so that differences between Version 1 and subsequent versions are resolved according to current VAX PASCAL definition.

3.2.2.14 /OPTIMIZE Qualifier

The /OPTIMIZE qualifier directs the compiler to optimize the code for the program or module being compiled so that more efficient code will be generated. A single identifier or a list of identifiers enclosed in parentheses may follow /OPTIMIZE; these identifiers are the names of options that tell the compiler which aspects of the compilation unit to optimize.

Table 3–4 lists the available options, their corresponding actions, and their negations.

Table 3–4: /OPTIMIZE Qualifier Options

Option	Action	Negation
ALL	Enables all optimization components	NONE
INLINE	Enables inline expansion of user-defined routines	NOINLINE

By default, /OPTIMIZE=ALL is enabled. The /OPTIMIZE qualifier without options is equivalent to /OPTIMIZE=ALL. The negation /NOOPTIMIZE is equivalent to /OPTIMIZE=NONE.

Some VAX PASCAL programs that run under VAX PASCAL Version 1, however, may require the use of the /NOOPTIMIZE qualifier in order to execute the same way with subsequent versions.

You should also consider using /NOOPTIMIZE when you are using either /DEBUG or /CHECK. Allowing optimizations to occur may make debugging difficult and may obscure some sections of the compilation unit that you would like to check. (See Chapter 4 and the *VAX PASCAL Reference Manual* for details.)

The OPTIMIZE and NOOPTIMIZE attributes in the source program or module override the /OPTIMIZE and /NOOPTIMIZE qualifiers in the command line (see Section 3.2.5.4).

3.2.2.15 /SHOW Qualifier

The /SHOW qualifier specifies a list of items to be included in the listing file.

A single identifier or a list of identifiers enclosed in parentheses may follow /SHOW; these identifiers are the names of options that inform the compiler which type of information it should generate. Table 3–5 lists the available options, their corresponding actions, and their negations.

Table 3–5: /SHOW Qualifier Options

Option	Action	Negation
ALL	Enables listing of all options	NONE
DICTIONARY	Enables listing of %DICTIONARY records	NODICTIONARY
HEADER	Enables page headers	NOHEADER
INCLUDE	Enables listing of %INCLUDE files	NOINCLUDE
INLINE	Enables listing of inline summary listing	NOINLINE
SOURCE	Enables listing of VAX PASCAL source code	NOSOURCE
STATISTICS	Enables listing of compilation statistics	NOSTATISTICS
TABLE_OF_ CONTENTS	Enables listing of a table of contents if the %TITLE or %SUBTITLE directive was specified	NOTABLE_OF_ CONTENTS

The inline summary enabled by the /SHOW=INLINE qualifier shows only the names of routines. If you want to know why routines were not expanded inline, you must use either the /OPTIMIZE=INLINE or the /OPTIMIZE=ALL qualifier.

The compiler ignores the /SHOW qualifier if the /LIST qualifier is not enabled. The default for the /SHOW qualifier is /SHOW=(DICTIONARY, HEADER, INCLUDE, SOURCE, STATISTICS, TABLE_OF_CONTENTS). Note that the TABLE_OF_CONTENTS option is enabled by default, but only affects the listing when either the %TITLE directive or the %SUBTITLE directive has been used within the compilation unit. The negation /NOSHOW is equivalent to /SHOW=NONE.

3.2.2.16 /STANDARD Qualifier

The /STANDARD qualifier causes the compiler to generate messages wherever the compilation unit uses nonstandard Pascal features. Standard features fall under either the ISO (ISO 7185-1983(E)) or the ANSI (ANSI/IEEE770X3.97-1983 American National Standards Institute, Inc.) standard. For the purposes of this chapter, the ISO and ANSI standards are referred to as the Pascal "standard". Table 3-6 lists the available options, their corresponding actions, and their negations.

Table 3-6: /STANDARD Qualifier Options

Option	Action	Negation
NONE	Disable standards checking	N.A.
ANSI	Use the rules of the ANSI standard	N.A.
ISO	Use the rules of the ISO standard	N.A.
VALIDATION	Perform validation for the given standard	NOVALIDATION

Note that the /STANDARD qualifier allows only two options to be used. The first option selects the standard to be used, ANSI or ISO. The second option determines whether the strict validation rules are to be enforced, [NO]VALIDATION. Therefore, /STANDARD=(ANSI, ISO, VALIDATION) is not allowed because both ANSI and ISO are specified.

By default, these information-level messages are written to the error file SYS$ERROR. (Nonstandard features are the language extensions incorporated in VAX PASCAL; see the *VAX PASCAL Reference Manual.* Note that using the VALIDATION option changes all nonstandard information-level messages to be error-level messages.

By default, /NOSTANDARD is enabled. The /STANDARD qualifier without options is equivalent to /STANDARD=(ANSI, NOVALIDATION). /STANDARD=VALIDATION is equivalent to /STANDARD=(ANSI, VALIDATION). The negation /NOSTANDARD is equivalent to /STANDARD=NONE.

3.2.2.17 /TERMINAL Qualifier

The /TERMINAL qualifier specifies a list of items to be displayed on the terminal.

A single identifier or a list of identifiers enclosed in parentheses may follow the /TERMINAL qualifier; these identifiers are options that inform the compiler which type of information to display. Table 3–7 lists the available options, their corresponding actions, and their negations.

Table 3–7: /TERMINAL Qualifier Options

Option	Action	Negation
ALL	Enables displaying of all options	NONE
FILE_NAME	Enables displaying of file names on PASCAL command line as they are being processed	NOFILE_NAME
ROUTINE_NAME	Enables displaying of routine names as code is generated	NOROUTINE_ NAME
STATISTICS	Enables displaying of compiler statistics	NOSTATISTICS

By default, /NOTERMINAL is enabled. The /TERMINAL qualifier without options is equivalent to /TERMINAL=ALL. The negation /NOTERMINAL is equivalent to /TERMINAL=NONE.

3.2.2.18 /USAGE Qualifier

The /USAGE qualifier directs the compiler to perform compile-time checks indicated by the chosen options.

A single identifier or a list of identifiers enclosed in parentheses may follow /USAGE; these identifiers are options that tell the compiler which checks to perform. Table 3–8 lists the available options, their corresponding actions, and their negations.

Table 3–8: /USAGE Qualifier Options

Option	Action	Negation
ALL	Enables checking of all options	NONE
UNCERTAIN	Checks for variables that may be uninitialized depending on program flow	NOUNCERTAIN
UNINITIALIZED	Checks for variables that are known to be uninitialized	NOUNINITIALIZED
UNUSED	Checks for variables that are declared but never referenced	NOUNUSED

The following types of variables are not checked for uninitialization:

- Variables that have a file component
- Predeclared INPUT or OUTPUT identifiers
- Variables that have global, external or inherited visibility
- Variables declared with the AT attribute
- Variables declared with the COMMON attribute
- Variables declared with the READONLY attribute
- Variables declared with the VOLATILE attribute
- Variables used as parameters
- Variables used as function identifiers

By detault, /USAGE=(UNINITIALIZED, NOUNUSED, NOUNCERTAIN) is enabled. The /USAGE qualifier without options is equivalent to /USAGE=ALL. The negation /NOUSAGE is equivalent to /USAGE=NONE.

3.2.2.19 /WARNINGS Qualifier

The /WARNINGS qualifier directs the compiler to generate diagnostic messages in response to warning-level or informational-level errors.

By default, these messages are written to the error file SYS$ERROR. A warning or informational 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.

Note that informational messages generated when the /STANDARD qualifier is enabled do not appear if /NOWARNINGS is enabled.

By default, /WARNINGS is enabled. The *VAX PASCAL Reference Manual* lists the compiler diagnostic messages.

3.2.3 Specifying Attributes in the Compilation Unit

Some of the features available with the PASCAL qualifiers can alternatively be specified in the source program or module by attributes. This section describes the relationship between attributes and PASCAL command qualifiers that perform the same actions. The *VAX PASCAL Reference Manual* contains complete information on all the VAX PASCAL attributes.

3.2.3.1 CHECK Attribute

The CHECK attribute enables error checking of routines and compilation units. CHECK has the following form:

```
CHECK [[ ({option},...) ]]
```

The available options are identical to those of the /CHECK command-line qualifier: ALL, BOUNDS, CASE_SELECTORS, OVERFLOW, POINTERS, SUBRANGE, and their negations. As with the /CHECK qualifier, only the BOUNDS option is enabled by default when compilation begins.

By using the CHECK attribute in the source program or module, you can select routines that should be checked for certain conditions. For example, you can check only pointer references and integer overflow in one routine, and then check only case-selector values and assignments to variables of subrange types in another.

With the /CHECK qualifier, however, you are restricted to checking the entire source program or module for the same specified or default options.

The CHECK attribute and any accompanying options take precedence over the /CHECK qualifier and any options specified with it.

3.2.3.2 ENVIRONMENT Attribute

The ENVIRONMENT attribute causes the creation of an environment file in which declarations and definitions made at the outermost level of a compilation unit are saved. It has the following form:

```
ENVIRONMENT(name-string)
```

You can name the environment file by specifying a VAX/VMS file description in the name string following the attribute. (A name string is a sequence of characters enclosed in apostrophes; it cannot use the extended string syntax.) The default file type is PEN, an abbreviation for "PASCAL Environment."

As with the /ENVIRONMENT qualifier, the ENVIRONMENT attribute applies only to compilation units and not to routines.

The /ENVIRONMENT qualifier overrides the ENVIRONMENT attribute.

3.2.3.3 G_FLOATING Attribute

The G_FLOATING attribute indicates that G_floating data and instructions should be used in the entire compilation unit for all operations involving double-precision values. The negation of this attribute, NOG_FLOATING, specifies the use of the D_floating representation and instructions.

As with the /G_FLOATING qualifier, the G_FLOATING attribute applies only to compilation units and not to routines.

VAX PASCAL generates a warning if an attribute specifying the hardware representation of double-precision values conflicts with the corresponding command-line qualifier.

3.2.3.4 OPTIMIZE Attribute

The OPTIMIZE attribute directs the compiler to optimize the code for routines and compilation units. OPTIMIZE has the following form:

```
OPTIMIZE[[ ({option},...) ]]
```

The available options are identical to those of the /OPTIMIZE command line qualifier: ALL, INLINE, and their negations. As with the /OPTIMIZE qualifier, /OPTIMIZE=ALL is enabled by default when compilation begins. The NOOPTIMIZE attribute disables optimization for a particular routine or compilation unit.

The VAX PASCAL compiler optimizes code by default, but if some sections of your compilation unit do not conform to the Pascal standard, optimization of these sections may cause unexpected results when you run the program. You may thus selectively enable and disable optimization depending on the requirements of a routine or compilation unit. The /OPTIMIZE and /NOOPTIMIZE qualifiers, however, control whether or not the code for an entire source program or module is optimized.

The optimization attributes take precedence over the /OPTIMIZE and /NOOPTIMIZE command-line qualifiers.

3.2.4 Specifying Output Files

The compiler produces listing files, object files, and environment files as output. You can control the production of these files by using the /LIST, /OBJECT, and /ENVIRONMENT qualifiers with the PASCAL command. The VAX PASCAL compiler generates output files according to the following rules:

- If you specify one source file, one output file is generated.
- If you separate multiple source files with commas, one output file is generated for each source file.
- If you separate multiple source files with plus signs, the files are concatenated, and one output file is generated.
- When you include a qualifier in a list of concatenated files, the qualifier affects all files in the list.
- You can use both plus signs and commas in the same command line to produce different combinations of concatenated and separate output files.

Example 1

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

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

Example 2

```
$ PASCAL X + Y + Z
```

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

Example 3

```
$ PASCAL/OBJECT=SQUARE
_File:  CIRCLE
```

The file CIRCLE.PAS is compiled, producing an object file named SQUARE.OBJ, but no listing file. (This example applies to interactive mode only.)

Example 4

```
$ PASCAL A + B/LIST, C
```

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

Example 5

```
$ PASCAL A + CIRC/NOOBJECT + X
```

The command shown above completely suppresses the object file; that is, source files A.PAS, CIRC.PAS, and X.PAS are concatenated and compiled, but no object file is produced.

Example 6

```
$ PASCAL/LIST [DIR]M
```

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

3.2.5 Compiler Listing Example

A complete compiler listing has six major parts:

- A table of contents listing, generated by the /SHOW=TABLE_OF_CONTENTS qualifier
- A source code listing, generated by the /LIST qualifier
- A cross-reference listing, generated by the /CROSS_REFERENCE qualifier (if a listing is being generated)
- A machine code listing, generated by the /MACHINE_CODE qualifier (if a listing is being generated)
- An inline summary listing, generated by the /SHOW=INLINE qualifier
- Pascal compilation statistics, generated by the /SHOW=STATISTICS qualifier

Figure 3–1 is a complete compiler listing. The circled numbers in the listing correspond to the numbered examples and explanations in Section 3.2.8.

Figure 3–1: VAX PASCAL Compiler Listing

```
❶                                                      ❷                    ❸               ❹
EXAMPLE                                               13-Aug-1986 15:40:38  VAX Pascal V3.5-1        Page    1
V1.0-1                    Table of Contents           13-Aug-1986 15:39:15  DISK$USER:[SMITH]SAMPLE.PAS;10 (1)
  ❺                         ❻                           ❼                                    ❽      ❾

     ❿      ⓫      ⓬
     Listing  Line    Source          ⓭
     Page     Number  Page          Section

        1        1    (1)        VAX PASCAL listing features
        1        2    (1)          Comments and declarations
        2       29    (2)          Main program body

EXAMPLE                   VAX PASCAL listing features  13-Aug-1986 15:40:38  VAX Pascal V3.5-1        Page    1
V1.0-1                    Comments and declarations    13-Aug-1986 15:39:15  DISK$USER:[SMITH]SAMPLE.PAS;10 (1)

⓮ ⓯ ⓰ ⓱
-LINE-IDC-PL-SL-

00001      0  0 %TITLE 'VAX PASCAL listing features'
                1
%PASCAL-I-STDCMPDIR, (1) Nonstandard: compiler directive
00002      0  0 %SUBTITLE 'Comments and declarations'
                1
%PASCAL-I-STDCMPDIR, (1) Nonstandard: compiler directive
00003      0  0
00004      0  0 [INHERIT('SYS$LIBRARY:STARLET'),IDENT('v1.0-1')] PROGRAM Example(INPUT,OUTPUT);
                1
%PASCAL-I-STDATTRLST, (1) Nonstandard: attribute list
00005      0  0
00006    C 0  0 { INCLUDE Declarations }
00007      0  0 %INCLUDE 'Testinc.PAS'
                1 ⓲
%PASCAL-I-STDINCLUDE, (1) Nonstandard: %INCLUDE directive
           ⓳         ⓴        ㉑
00008 I    0  0 TYPE
00009 I    0  0    Arrtype = ARRAY [1..5] OF INTEGER;
00010 I    0  0 VAR
00011 I    0  0    Arr : Arrtype;
00012 I    0  0    I,Temp : INTEGER;
00013 I    0  0
```

(Continued on next page)

Figure 3–1 (Cont.): VAX PASCAL Compiler Listing

```
00014 I    1  0 PROCEDURE Fill ( VAR Arr : Arrtype );
00015 I    1  0     VAR I : INTEGER;
00016 I    1  1     BEGIN
00017 I    1  1     FOR I := 1 TO 5 DO
00018 I    1  1         Arr[I] := I;
00019 I    0  0     END;
00020 I    0  0
00021 I    1  0 [OPTIMIZE(NOINLINE)] PROCEDURE Print ( Arr : Arrtype );
                1
%PASCAL-I-STDATTRLST, (1) Nonstandard: attribute list
00022 I    1  0     VAR I : INTEGER;
00023 I    1  1     BEGIN
00024 I    1  1     FOR I := 1 TO 5 DO
00025 I    1  1       WRITELN('The ',i:2,'th element is ',arr[i]:2 );
00026 I    0  0     END;
00027      0  0

EXAMPLE                       VAX PASCAL listing features    13-Aug-1986 15:40:38   VAX Pascal V3.5-1          Page    2
V1.0-1                        Main program body              13-Aug-1986 15:39:15   DISK$USER:[SMITH]SAMPLE.PAS;10 (2)

-LINE-IDC-PL-SL-

00029      0  0 %subtitle 'Main program body'
                1
%PASCAL-I-STDCMPDIR, (1) Nonstandard: compiler directive
00030      0  0
00031      0  1 BEGIN
00032      0  1 Fill( Arr );
00033      0  1
00034      0  1 FOR I := 2 TO 5 DO
00035      0  2     BEGIN
00036      0  2     Temp := Arr[I];
00037      0  2     Arr[I] := Arr[I-1];
00038      0  2     Arr[I-1] := Temp;
00039      0  1     END;
00040      0  1
00041      0  1 Print( Arr );
00042      0  0 END.
```

(Continued on next page)

Figure 3–1 (Cont.): VAX PASCAL Compiler Listing

```
EXAMPLE                         VAX PASCAL listing features     13-Aug-1986 15:40:38    VAX Pascal V3.5-1           Page    3
V1.0-1                          Cross Reference Listing         13-Aug-1986 15:39:15    DISK$USER:[SMITH]SAMPLE.PAS;10 (2)
(22)            (23)
ARR             VAR    ARRTYPE { IN PROGRAM EXAMPLE }
                       (24) 11      32 P    36      36      37 =    37      37      38 =    41
ARR             PARM   ARRTYPE { IN PROCEDURE FILL }
                            14      18 =
ARR             PARM   ARRTYPE { IN PROCEDURE PRINT }
                            21      25
ARRTYPE         TYPE   ARRAY OF INTEGER { IN PROGRAM EXAMPLE }
                            9       11      14      21
EXAMPLE         PROG   [PSECT($CODE)]
                            4
FILL            PROC   [PSECT($CODE), UNBOUND]  { IN PROGRAM EXAMPLE }
                            14      32
I               VAR    INTEGER { IN PROGRAM EXAMPLE }
                            12      34 =    36      37      37      38
I               VAR    INTEGER { IN PROCEDURE FILL }
                            15      17 =    18      18      18
I               VAR    INTEGER { IN PROCEDURE PRINT }
                            22      24 =    25      25
INPUT           VAR    [EXTERNAL(PAS$FV_INPUT)] TEXT { IN PROGRAM EXAMPLE } (25)
                            *       4
INTEGER         TYPE
                            *       9       12      15      22
OUTPUT          VAR    [EXTERNAL(PAS$FV_OUTPUT)] TEXT { IN PROGRAM EXAMPLE } (26)
                            *       4       25
PRINT           PROC   [PSECT($CODE), UNBOUND]  { IN PROGRAM EXAMPLE }
                            21      41
TEMP            VAR    INTEGER { IN PROGRAM EXAMPLE }
                            12      36 =    38      38
WRITELN         PROC   { BUILTIN } (27)
                            *       25


ENVIRONMENT FILE SUMMARY (28)
------------------------
 #1        SYS$COMMON:[SYSLIB]STARLET.PEN;18
```

(Continued on next page)

Figure 3-1 (Cont.): VAX PASCAL Compiler Listing

```
+-------------------------------------------------------------------+
| KEY TO REFERENCE FLAGS: 29                                        |
|                                                                   |
|       *  predeclared                                              |
|       =  modified                                                 |
|       A  address of                                               |
|       F  forward declared procedure or function                   |
|       L  label defining reference                                 |
|       P  passed as a parameter, possibly modified or called       |
|       R  READ operation done                                      |
|       W  used in a WITH statement                                 |
+-------------------------------------------------------------------+

EXAMPLE                 VAX PASCAL listing features    13-Aug-1986 15:40:38   VAX Pascal V3.5-1          Page    4
V1.0-1                  Generated Code                 13-Aug-1986 15:39:15   DISK$USER:[SMITH]SAMPLE.PAS;10 (2)
                                                                      .TITLE  EXAMPLE 30
                                                                      .IDENT  \V1.0-1\

                                                       00000          .PSECT  $CODE,PIC,CON,REL,LCL,SHR,EXE,RD,NOWRT,2 31
                                                          33
                                      20  65  68  54   00000 C.AAA:   .ASCII  \The \
 20  73  69  20  74  6E  65  6D  65  6C  65  20  68  74   00004 C.AAB:   .ASCII  \th element is \
                                                    01 00012          NOP                                      ; 0021
                                                    01 00013          NOP
                                                              34               35
                                                  003C 00000 PRINT:   .WORD   ^M<R2,R3,R4,R5>
                                    5E          1C  C2 00002          SUBL2   #28,SP
                                         F8     AD  D4 00005          CLRL    -8(FP)
                                    6D 00000000G EF  9E 00008          MOVAB   PAS$HANDLER,(FP)
                       E4   AD       04   BC    14  28 0000F          MOVC3   #20,@4(R12),ARR
                              32          5C    01  D0 00015 1$:     MOVL    #1,R12                           ; 0024
                                          52    5C  D0 00018 2$:     MOVL    R12,I
                                     FFFFFFCB   EF  9F 0001B          PUSHAB  C.AAA                            ; 0025
                                                04  DD 00021          PUSHL   #4
                                     00000000G  EF  9F 00023          PUSHAB  PAS$FV_OUTPUT
                          00000000G  EF         03  FB 00029          CALLS   #3,PAS$WRITE_STRING
                                                02  DD 00030          PUSHL   #2
                                                52  DD 00032          PUSHL   I
                                     00000000G  EF  9F 00034          PUSHAB  PAS$FV_OUTPUT
```

(Continued on next page)

Figure 3–1 (Cont.): VAX PASCAL Compiler Listing

```
                  00000000G  EF          03   FB 0003A           CALLS   #3,PAS$WRITE_INTEGER
                        FFFFFFA9   EF    9F 00041           PUSHAB  C.AAB
                                   0E    DD 00047           PUSHL   #14
                        00000000G  EF    9F 00049           PUSHAB  PAS$FV_OUTPUT
                  00000000G  EF          03   FB 0004F           CALLS   #3,PAS$WRITE_STRING
                                   02    DD 00056           PUSHL   #2
    01        05          01       52    0A 00058           INDEX   I,#1,#5,#1,#0,R0
                          50       00       0005D
                                                             39      40
                              E0 AD40    DD 0005F           PUSHL   ARR-4[R0]
                           38      37       36
                        00000000G  EF    9F 00063           PUSHAB  PAS$FV_OUTPUT
                  00000000G  EF          03   FB 00069           CALLS   #3,PAS$WRITE_INTEGER
                        00000000G  EF    9F 00070           PUSHAB  PAS$FV_OUTPUT
                  00000000G  EF          01   FB 00076           CALLS   #1,PAS$WRITELN2
              97          5C       05    F3 0007D           AOBLEQ  #5,R12,2$ 41                    42
                                         04 00081           RET                                   ; 0026

; Routine Size: 130 bytes,    Routine Base: $CODE + 00014
                                         01 00082           NOP                                   ; 0004
                                         01 00083           NOP

                                     000C 00000 EXAMPLE:.WORD   ^M<R2,R3>
                          5E       1C    C2 00002           SUBL2   #28,SP
                              F8   AD    D4 00005           CLRL    -8(FP)
                          6D 00000000G  EF    9E 00008           MOVAB   PAS$HANDLER,(FP)
                          50       01    D0 0000F 1$:       MOVL    #1,R0                         ; 0017
                                         01 00012           NOP
                                         01 00013           NOP
                          51       50    D0 00014 2$:       MOVL    R0,I
    01        05          01       51    0A 00017           INDEX   I,#1,#5,#1,#0,R12             ; 0018
                          5C       00       0001C
                     E0 AD4C       51    D0 0001E           MOVL    I,ARR-4[R12]
              ED          50       05    F3 00023           AOBLEQ  #5,R0,2$
```

(Continued on next page)

Figure 3-1 (Cont.): VAX PASCAL Compiler Listing

```
EXAMPLE                       VAX PASCAL listing features     13-Aug-1986 15:40:38    VAX Pascal V3.5-1            Page    5
V1.0-1                        Generated Code                  13-Aug-1986 15:39:15    DISK$USER:[SMITH]SAMPLE.PAS;10 (2)
                                       50             02  D0 00027         MOVL    #2,R0                                    ; 0034
                                                          01 0002A         NOP
                                                          01 0002B         NOP
                                       51             50  D0 0002C 3$:     MOVL    R0,I
           01            05            01             51  0A 0002F         INDEX   I,#1,#5,#1,#0,R2                         ; 0036
                                       52             00     00034
                                       52        E0 AD42  DE 00036         MOVAL   ARR-4[R2],R2
                                       5C             62  D0 0003B         MOVL    (R2),TEMP
                         53            51             01  C3 0003E         SUBL3   #1,I,R3                                  ; 0037
           01            05            01             53  0A 00042         INDEX   R3,#1,#5,#1,#0,R3
                                       53             00     00047
                                       53        E0 AD43  DE 00049         MOVAL   ARR-4[R3],R3
                                       62             63  D0 0004E         MOVL    (R3),(R2)
                                       63             5C  D0 00051         MOVL    TEMP,(R3)                                ; 0038
                         D4            50             05  F3 00054         AOBLEQ  #5,R0,3$
                                                  E4  AD  9F 00058         PUSHAB  ARR                                      ; 0041
                                0014   CF             01  FB 0005B         CALLS   #1,PRINT
                                       50             01  D0 00060         MOVL    #1,R0                                    ; 0042
                                                          04 00063         RET
           ㊸                          ㊹
; Routine Size: 100 bytes,    Routine Base: $CODE + 00098

                                                             000FC         .END
```

(Continued on next page)

Figure 3–1 (Cont.): VAX PASCAL Compiler Listing

```
EXAMPLE                 VAX PASCAL listing features     13-Aug-1986 15:40:38    VAX Pascal V3.5-1          Page   6
V1.0-1                  Inline Summary Listing          13-Aug-1986 15:39:15    DISK$USER:[SMITH]SAMPLE.PAS;10 (2)

 { IN PROCEDURE PRINT } (45)

 { IN PROGRAM EXAMPLE }

   FILL -> EXAMPLE at line 32 (46)
   PRINT <> EXAMPLE at line 41 (47)
      Inline optimization not enabled (48)


+---------------------------------------+
|  KEY TO REFERENCE FLAGS:  (49)        |
|                                       |
|        ->   expanded into             |
|        <>   not expanded into         |
+---------------------------------------+
EXAMPLE                 VAX PASCAL listing features     13-Aug-1986 15:40:38    VAX Pascal V3.5-1          Page   7
V1.0-1                  Pascal Compilation Statistics   13-Aug-1986 15:39:15    DISK$USER:[SMITH]SAMPLE.PAS;10 (2)

PSECT SUMMARY
      (50)                    (51)                         (52)
      Name                    Bytes                        Attributes

$CODE                             252  NOVEC,NOWRT, RD, EXE, SHR, LCL, REL, CON, PIC,ALIGN(2)

ENVIRONMENT STATISTICS
                                                             (54)
      (53)                                           -------- Symbols --------
      File                                           Total    Loaded   Percent

SYS$COMMON:[SYSLIB]STARLET.PEN;18                    20457        0         0
```

(Continued on next page)

Figure 3-1 (Cont.): VAX PASCAL Compiler Listing

```
COMMAND QUALIFIERS

55 PASCAL/MACH/LIS/SHOW=ALL/OPTIMIZE=ALL/STANDARD/CROSS LISTING

     /CHECK=(BOUNDS,NOCASE_SELECTORS,NOOVERFLOW,NOPOINTERS,NOSUBRANGE)
     /DEBUG=(NOSYMBOLS,TRACEBACK)
     /SHOW=(DICTIONARY,INCLUDE,INLINE,HEADER,SOURCE,STATISTICS,TABLE_OF_CONTENTS)
     /OPTIMIZE
 56  /STANDARD=(ANSI,NOVALIDATION)
     /TERMINAL=(NOFILE_NAME,NOROUTINE_NAME,NOSTATISTICS)
     /USAGE=(NOUNUSED,UNINITIALIZED,NOUNCERTAIN)
     /NOANALYSIS_DATA
     /NODIAGNOSTICS
     /NOENVIRONMENT
     /LIST=DISK$USER:[SMITH]SAMPLE.LIS;14
     /OBJECT=DISK$USER:[SMITH]SAMPLE.OBJ;13
     /CROSS_REFERENCE /ERROR_LIMIT=30 /NOG_FLOATING /MACHINE_CODE /NOOLD_VERSION /WARNINGS

COMPILER INTERNAL TIMING       57

  Phase                 Faults      CPU Time      Elapsed Time
  Initialization          203       00:00.5         00:01.5
  Source Analysis         987       00:05.2         00:09.9
  Source Listing          374       00:01.4         00:05.5
  Tree Construction       161       00:00.2         00:00.6
  Flow Analysis            50       00:00.1         00:00.6
  Value Propagation         9       00:00.0         00:00.0
  Profit Analysis          56       00:00.2         00:00.4
  Context Analysis         70       00:00.4         00:00.8
  Name Packing              5       00:00.0         00:00.2
  Code Selection           58       00:00.1         00:00.3
  Final                   160       00:00.6         00:02.1
  TOTAL                  2149       00:08.8         00:22.1
```

(Continued on next page)

Figure 3–1 (Cont.): VAX PASCAL Compiler Listing

```
COMPILATION STATISTICS      18

  Informationals: 6
  Warnings:       0
  Errors:         0
  Fatals:         0
  CPU Time:       00:08.8        (287 Lines/Minute)

EXAMPLE                         VAX PASCAL listing features     13-Aug-1986 15:40:38    VAX Pascal V3.5-1           Page    8
V1.0-1                          Pascal Compilation Statistics   13-Aug-1986 15:39:15    DISK$USER:[SMITH]SAMPLE.PAS;10 (2)

  Elapsed Time:   00:22.1
  Page Faults:    2149
  Pages Used:     1949
  Compilation Complete
```

3.2.6 Compiler Listing Format

This section explains the format of the compiler listing and describes each part in detail. In the examples of each part, significant items are marked with circled numbers that correspond to the explanations following the examples. The numbers in Section 3.2.7 correspond with the numbers in this section.

Each page of the compiler listing begins with a title line and a source file line:

```
❶                                     ❷                                   ❸                          ❹
EXAMPLE                    13-Aug-1986 15:40:38   VAX PASCAL V3.5-1            Page  1
```

The title line has four parts:

❶ The module name, which corresponds to the name in the program or module heading.

❷ The date (day, month, year) and time (hour, minute, second) of compilation.

❸ The VAX PASCAL compiler name and version number.

❹ The page number.

```
  ❺       ❻                    ❼                           ❽                          ❾
V1.0-1  Table of Contents   13-Aug-1986 15:39:15  DISK$USER:[SMITH]SAMPLE.PAS;1 (1)
```

The source file information consists of five parts:

❺ The module identifier, which is either specified by the IDENT attribute or is the default 01.

❻ The subtitle that describes which part of the complete listing is found on this page.

❼ The date (day, month, year) and time (hour, minute, second) of source file creation.

❽ The VAX/VMS file specification of the source file that is being compiled when the compiler directs output to a new page of the listing.

❾ The page number of the source file.

3.2.7 Table of Contents Listing

The table of contents listing, generated by the /SHOW=TABLE_OF_CONTENTS and /LIST qualifiers, appears first in the generated list. Note that even though /SHOW=TABLE_OF_CONTENTS is enabled by default, the table of contents is generated only when either a %TITLE or %SUBTITLE appears in the compilation unit.

```
 ⑩       ⑪        ⑫
Listing Line     Source            ⑬
Page    Number   Page             Section

    1        1    (1)         VAX PASCAL listing features
    1        2    (1)             Comments and declarations
    2       29    (2)             Main program body
```

⑩ The page number of the source listing for the section.

⑪ The line number where the section begins.

⑫ The page number of the source file for the section.

⑬ The name of the section.

3.2.8 Source Code Listing

The source code is printed in the source code listing with numbers that correspond to each line.

```
⑭     ⑮ ⑯ ⑰
-LINE-IDC-PL-SL-

00001       0  0 %title 'VAX PASCAL listing features'
                 1
%PASCAL-I-STDCMPDIR, (1) Nonstandard: compiler directive
00002       0  0 %subtitle 'Comments and declarations'
                 1
%PASCAL-I-STDCMPDIR, (1) Nonstandard: compiler directive
00003       0  0
00004       0  0 [inherit('sys$library:starlet'),ident('V1.0-1')] program example(input,output);
                 1
%PASCAL-I-STDATTRLST, (1) Nonstandard: attribute list
00005       0  0
00006    C  0  0 { include declarations }
```

⑭ Source code line numbers—The compiler assigns unique line numbers to the lines of source code in a VAX PASCAL compilation unit. These line numbers appear in the leftmost column of the source code listing. The symbolic traceback that is printed if your program encounters an error at run time refers to these line numbers; in addition, the

VAX/VMS Debugger uses these line numbers when controlling program execution.

⓯ Listing flags—The compiler records additional information about each source line. Each source line can have any or all of the following qualities:

- I—The source line came from a %INCLUDE directive.
- D—The source line came from a %DICTIONARY directive.
- C—The source line contains nothing but comments and blank space.

⓰ Procedure nesting level—The compiler keeps track of the declaration depth of PROCEDURES and FUNCTIONS. The level is incremented every time a PROCEDURE or FUNCTION reserved word occurs and is decremented when the end of the block is reached. The procedure level reflects the level that is active at the end of the source line.

⓱ Statement nesting level—The compiler keeps track of the nesting depth of structured statements. The level is incremented every time a BEGIN, REPEAT, or CASE statement is seen. The level is decremented when the end of a structured statement is seen. The statement level reflects the level that is active at the end of the source line.

Diagnostic Messages

The source code listing includes errors, warnings, and informational messages produced by the compiler. The lines beneath the source code line in which the error is reported specify which kind of message was generated. For example:

```
00007    0 0 %INCLUDE 'Testinc.PAS'
               1 ⓲
%PASCAL-I-STDINCLUDE, (1) Nonstandard: %INCLUDE directive
      ⓳             ⓴       ㉑
```

Diagnostic messages include the following information:

⓲ A digit or letter that points to the position on the line where the error probably occurred.

⓳ A code that indicates the name of the compiler (PASCAL), the severity of the error (either I (INFORMATIONAL), W (WARNING), E (ERROR), or F (FATAL)), and the name of the error message generated.

⓴ A digit or letter in parentheses that corresponds to the digit or letter that identifies the position where the error was detected.

㉑ The text that corresponds to that of each error code.

Note that one source code error often causes the compiler to detect more errors at the same position.

3.2.9 Cross-Reference Listing

The cross-reference listing, generated by the /CROSS_REFERENCE and /LIST qualifiers, follows the source code listing. The listing begins with a title line and the subtitle Cross Reference Listing. The format of this line is identical to the format of the title line in the source code listing.

The entries in the cross-reference listing are the names of every identifier, both predeclared and user-declared, and every label to which the source code refers.

```
(22)           (23)
ARR            VAR    ARRTYPE { IN PROGRAM EXAMPLE }
                      (24) 11      32 P     36       36       37 =     37       37       38 =     41
ARR            PARM   ARRTYPE { IN PROCEDURE FILL }
                            14       18 =
ARR            PARM   ARRTYPE { IN PROCEDURE PRINT }
                            21       25
ARRTYPE        TYPE   ARRAY OF INTEGER { IN PROGRAM EXAMPLE }
                             9       11       14       21
EXAMPLE        PROG   [PSECT($CODE)]     (25)
                             4
FILL           PROC   [PSECT($CODE), UNBOUND]  { IN PROGRAM EXAMPLE }
                            14       32
I              VAR    INTEGER { IN PROGRAM EXAMPLE }
                            12       34 =     36       37       37       38
I              VAR    INTEGER { IN PROCEDURE FILL }
                            15       17 =     18       18       18
I              VAR    INTEGER { IN PROCEDURE PRINT }
                            22       24 =     25       25
INPUT          VAR    [EXTERNAL(PAS$FV_INPUT)] TEXT { IN PROGRAM EXAMPLE }
                             *        4
INTEGER        TYPE
                             *        9       12       15       22
OUTPUT         VAR    [EXTERNAL(PAS$FV_OUTPUT)] TEXT { IN PROGRAM EXAMPLE } (26)
                             *        4       25
PRINT          PROC   [PSECT($CODE), UNBOUND]  { IN PROGRAM EXAMPLE }
                            21       41
TEMP           VAR    INTEGER { IN PROGRAM EXAMPLE }
                            12       36 =     38       38
WRITELN        PROC   { BUILTIN } (27)
                             *       25

ENVIRONMENT FILE SUMMARY (28)
------------------------
 #1       SYS$COMMON:[SYSLIB]STARLET.PEN;18
```

```
+--------------------------------------------------------------+
| KEY TO REFERENCE FLAGS:  ㉙                                   |
|                                                              |
|       *  predeclared                                         |
|       =  modified                                            |
|       A  address of                                          |
|       F  forward declared procedure or function              |
|       L  label defining reference                            |
|       P  passed as a parameter, possibly modified or called  |
|       R  READ operation done                                 |
|       W  used in a WITH statement                            |
+--------------------------------------------------------------+
```

The cross-reference listing contains the following information:

㉒ Labels and identifiers, listed in alphabetical order.

㉓ An abbreviation for the program element represented by a label or an identifier. Program elements can be labels, symbolic constants, types (predefined and user-defined), variables, field identifiers, procedures and functions (predeclared and user-written), formal parameters, programs, and modules.

㉔ The line numbers where each identifier or label is used. The first line number indicates where the label or identifier is declared. (Note that the declarations of predeclared identifiers are not listed.) Subsequent references to labels and identifiers are noted by abbreviations that describe the specific use.

㉕ For executable blocks, a list of the attributes associated with the block, and the name of the declaring block enclosed in braces ({}). If the block is part of a function, the function result type is listed.

㉖ For variables, a list of the attributes associated with the variable, an identifier or description that designates the variable's type, and the name of the block in which the variable was declared. If the variable is a record field, the name of the record type (if it has a name) is listed in braces.

㉗ For predeclared routines, the notation { BUILTIN }. If the routine is a function and its result type can be determined at compile time, the type is listed.

㉘ A list of all environment files inherited by the compilation unit.

㉙ A key to the abbreviations that indicate how an identifier or a label is used.

3.2.10 Machine Code Listing

The machine code listing, generated by the /MACHINE_CODE and /LIST qualifiers, follows the cross-reference listing. Note that the VAX PASCAL compiler does not generate machine code for a source program if error messages are produced at compile time. However, informational and warning messages do not interfere with the compiler's generation of machine code.

The machine code listing begins with a title line and the subtitle Generated Code. The format of this line is identical to the format of the title lines in the source code and cross-reference listings.

The following examples illustrate the kinds of information contained in the machine code listing.

Title

```
.TITLE  EXAMPLE    ㉚
.IDENT  \V1.0-1\
```

㉚ The object module title and ident fields. The object module title is the same as the name specified in the program or module heading. The ident field is derived from the IDENT attribute or is the default 01.

PSECT Information

```
.PSECT   $CODE,PIC,CON,REL,LCL,SHR,EXE,RD,NOWRT,2     ㉛
                 .
                 .
                 .
```

㉛ Information about the program sections in which storage has been allocated (except program sections containing automatic variables). The name of each program section is followed by a list of its properties and then its contents.

Machine Instructions

```
                                                           ㉜
         52              5C   D0 00018 2$:      MOVL     R12,I
            FFFFFFCB     EF   9F 0001B          PUSHAB   C.AAA                          ; 0025
                         04   DD 00021          PUSHL    #4
            00000000G    EF   9F 00023          PUSHAB   PAS$FV_OUTPUT
00000000G   EF           03   FB 00029          CALLS    #3,PAS$WRITE_STRING
                         02   DD 00030          PUSHL    #2
                         52   DD 00032          PUSHL    I
            00000000G    EF   9F 00034          PUSHAB   PAS$FV_OUTPUT
00000000G   EF           03   FB 0003A          CALLS    #3,PAS$WRITE_INTEGER
            FFFFFFA9     EF   9F 00041          PUSHAB   C.AAB
                         0E   DD 00047          PUSHL    #14
            00000000G    EF   9F 00049          PUSHAB   PAS$FV_OUTPUT
00000000G   EF           03   FB 0004F          CALLS    #3,PAS$WRITE_STRING
```

㉜ The contents of a program section that contains machine instructions; note that the formats of these instructions are similar (but not identical) to the listings of VAX MACRO source code.

Literal Constants

```
           ㉝
00000   C.AAA:    .ASCII    \The \
00004   C.AAB:    .ASCII    \th element is\
```

㉝ Names of the form C.AAA, C.AAB, and so forth, that the compiler generates for literal constants such as character-string prompts.

Entry Point and Save Mask Definition

```
                 ㉞           ㉟
003C 00000  PRINT: .WORD    ^M<R2,R3,R4,R5>
```

For each executable block:

㉞ An entry point; if the block is global, the entry point is specified by .ENTRY; otherwise, the entry point is specified by the program or routine name and .WORD.

㉟ A register save mask definition.

Sides of Listing

```
                    E0 AD40   DD 0005F
                  ㊳     ㊲       ㊱
               00000000G  EF   9F 00063
00000000G  EF             03   FB 00069
               00000000G  EF   9F 00070
00000000G  EF             01   FB 00076
97         5C             05   F3 0007D
```

The left side of the listing contains the following:

㊱ The current location counter value. It appears in the center of the listing and divides the listing into halves. The location counter for routines is expressed as an offset from the beginning of the routine. To compute the absolute offset of a routine from the beginning of the program section, you add the location counter to the routine base (described later). The location counter for data is expressed as an absolute offset from the beginning of the program section.

㊲ The hexadecimal representation of the code, in right-to-left order.

㊳ Operands marked with the letter G, which will be modified by the linker. Such operands refer to symbols that are stored in a program section other than the current one or found in external routines.

```
㊴         ㊵
PUSHL      ARR-4[R0]
PUSHAB     PAS$FV_OUTPUT
CALLS      #3,PAS$WRITE_INTEGER
PUSHAB     PAS$FV_OUTPUT
CALLS      #1,PAS$WRITELN2
AOBLEQ     #5,R12,2$ ㊶
RET                                            ; 0026 ㊷
```

The right side of the listing contains the following:

㊴ Compiler-generated labels, designated by an integer followed by a dollar sign ($). They are used as the targets of compiler-generated instructions.

㊵ The symbolic op code.

㊶ A set of symbolic operands (if needed).

㊷ A comment section that can follow the symbolic operand; it contains a source line number if the corresponding instruction is the first one generated for that source line.

Summary Line

A summary line at the end of each executable block contains the following information: A

```
          ㊸                          ㊹
Routine Size: 100 bytes,     Routine Base: $CODE + 00098
```

㊸ The routine size in bytes.

㊹ The routine base, expressed as an offset from the start of the program section in which storage is allocated for the block.

3.2.11 Inline Summary Listing

The inline summary listing enables a user to determine which routine calls of user-defined routines were or were not expanded inline. The entries in the listing are the names of every routine call of user-defined routines.

You generate the inline summary listing by specifying the /SHOW=INLINE (see the /SNOW and /LIST qualifiers, Sections 3.2.4.15 and 3.2.4.10). The listing follows the source code listing or the cross-reference listing, if one is being generated. It begins with a title line and the subtitle Inline Summary Listing; the format of this line is identical to the format of the title line in the source code listing.

```
 { IN PROCEDURE PRINT } ㊺

 { IN PROGRAM EXAMPLE }
    FILL -> EXAMPLE at line 32 ㊻
    PRINT <> EXAMPLE at line 41 ㊼
        Inline optimization not enabled ㊽


+---------------------------------------+
|   KEY TO REFERENCE FLAGS:     ㊾       |
|                                       |
|         ->   expanded into            |
|         <>   not expanded into        |
+---------------------------------------+
```

The inline summary listing contains the following information:

㊺ The name of the routine, program, or module which contains calls to user-defined routines.

㊻ The call to the user-defined routine which was expanded, followed by the listing line number of the call.

47 The call to a user-defined routine which was not expanded, followed by the listing line number of the call.

48 In order for the list of reasons to be generated, the compilation unit must have been compiled with the inline summary listing enabled as well as inline expansion optimization specifically enabled by either /OPTIMIZE=INLINE or /OPTIMIZE=ALL. Note that although /OPTIMIZE defaults to /OPTIMIZE=ALL, you must explicitly specify the ALL option to generate these reasons.

49 A key to the abbreviations that indicate whether or not inline code expansion of a routine occurred.

3.2.12 Compilation Statistics

On the last page of the listing, the compiler prints the following categories of summary information:

- Statistics of all program sections created during compilation
- Statistics of all environment files inherited by the compilation
- The exact command line passed by DCL to the VAX PASCAL compiler
- A list of the command qualifier options in effect during compilation
- Statistics regarding the internal timing of the compiler
- Error and page fault counts and timing totals

The /LIST qualifier causes this information to be printed on the last page of a source code listing, regardless of whether cross-reference and machine code listings are also generated.

This page begins with a title line and the subtitle Pascal Compilation Statistics. The format of this line is identical to the format of the title lines in the source code and cross-reference listings.

```
PSECT SUMMARY
      50                            51                   52
    Name                          Bytes               Attributes

$CODE                                252  NOVEC,NOWRT, RD, EXE, SHR, LCL, REL, CON, PIC,ALIGN(2)

ENVIRONMENT STATISTICS
                                                        54
      53                                        -------- Symbols --------
    File                                        Total    Loaded   Percent

SYS$COMMON:[SYSLIB]STARLET.PEN;18               20457      0         0
```

```
COMMAND QUALIFIERS
55  PASCAL/MACH/LIST/SHOW=ALL/OPTIMIZE=ALL/STANDARD/CROSS TESTPROG

      /CHECK=(BOUNDS,NOCASE_SELECTORS,NOOVERFLOW,NOPOINTERS,NOSUBRANGE)
      /DEBUG=(NOSYMBOLS,TRACEBACK)
      /SHOW=(DICTIONARY,INCLUDE,INLINE,HEADER,SOURCE,STATISTICS,TABLE_OF_CONTENTS)
      /OPTIMIZE
      /STANDARD=(ANSI,NOVALIDATION)
56    /TERMINAL=(NOFILE_NAME,NOROUTINE_NAME,NOSTATISTICS)
      /USAGE=(NOUNUSED,UNINITIALIZED,NOUNCERTAIN)
      /NOENVIRONMENT
      /LIST=DISK$USER:[SMITH]SAMPLE.LIS;14
      /OBJECT=DISK$USER:[SMITH]SAMPLE.OBJ;13
      /CROSS_REFERENCE /ERROR_LIMIT=30 /NOG_FLOATING /MACHINE_CODE /NOOLD_VERSION /WARNINGS
```

50 The names of all program sections created by the compilation.

51 The total allocation in bytes for each program section.

52 The attributes of each program section

53 The names of each environment file inherited by the compilation

54 The total number of symbols in the environment file, followed by the number actually used by the compilation, followed by an integer percentage of used symbols versus defined symbols. (Note that the VAX PASCAL compiler's definition of "symbols" is in terms of internal representation, and may not reflect the complexity of the environment source; that is, the number of symbols shown loaded may not reflect the number of symbols in your program.)

55 The PASCAL command and its qualifiers. Note that if you abbreviate the qualifiers, they are printed in the abbreviated form.

56 The state of the compile-time qualifiers and any options.

Compiler Internal Timing Statistics

```
                              57
COMPILER INTERNAL TIMING

  Phase                 Faults      CPU Time      Elapsed Time
  Initialization          203       00:00.5         00:01.5
  Source Analysis         987       00:05.2         00:09.9
  Source Listing          374       00:01.4         00:05.5
  Tree Construction       161       00:00.2         00:00.6
  Flow Analysis            50       00:00.1         00:00.6
  Value Propagation         9       00:00.0         00:00.0
  Profit Analysis          56       00:00.2         00:00.4
  Context Analysis         70       00:00.4         00:00.8
  Name Packing              5       00:00.0         00:00.2
  Code Selection           58       00:00.1         00:00.3
  Final                   160       00:00.6         00:02.1
  TOTAL                  2149       00:08.8         00:22.1
```

㊼ The internal timing statistics record the number of page faults that occurred, and the amount of elapsed time and CPU time required for each phase of compilation.

Compilation Statistics Summary

```
                           ㊽
COMPILATION STATISTICS

  Informationals: 6
  Warnings:       0
  Errors:         0
  Fatals:         0
  CPU Time:       00:08.8         (287 Lines/Minute)
  Elapsed Time:   00:22.1
  Page Faults:    2149
  Pages Used:     1949
  Compilation Complete
```

㊽ The summary information includes the total number of messages generated at each level—information, warning, error, and fatal; the amount of time and speed of compilation, and the number of page faults that occurred. If the source code compiled without error, the error levels are not listed. The last line is a message indicating that the compilation of the source code is complete.

3.3 Linking a VAX PASCAL Program

Once you have compiled a VAX PASCAL source program or module, link it by using the DCL command LINK. The LINK command combines your object modules into one executable image, which can then be executed by the VAX/VMS system. A source program or module cannot run on the system until it is linked.

When you execute the LINK command, the VAX/VMS Linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references
- Signals an error message for any unresolved symbolic reference
- Allocates virtual memory space for the executable image

When using the LINK command on development systems, you may want to use the /DEBUG qualifier when you link your program module. The /DEBUG qualifier appends to the image all the symbol and line number information appended to the object modules plus information on global symbols, and forces the image to run under debugger control when it is executed.

The LINK command produces an executable image by default. However, you can also use the LINK command to obtain shareable images. The /SHAREABLE qualifier directs the linker to produce a shareable image. See Section 3.3.2 for a complete description of these and other LINK command qualifiers.

For a complete discussion of the VAX/VMS Linker, see the *VAX/VMS Linker Reference Manual.*

3.3.1 LINK Command

The LINK command has the following format:

```
                                                                {+}
LINK⟦{/command-qualifier}...⟧ {file-spec⟦{/file-qualifier}...⟧} { } ...
                                                                {,}
```

/command-qualifier
The name of a qualifier (see Section 3.2.2) that indicates special processing to be performed by the linker on all files listed.

file-spec
The name of an object file that contains the object code to be linked. If you separate multiple file specifications with commas, the programs are linked separately. If you separate the file specifications with plus signs, the files are concatenated and linked as one program.

/file-qualifier
The name of a qualifier (see Section 3.2.2) that indicates special processing to be performed by the linker on the files to which the qualifier is attached.

The following command line links the object files DANCE.OBJ, CHACHA.OBJ, and SWING.OBJ to produce one executable image called DANCE.EXE:

```
$ LINK DANCE.OBJ, CHACHA.OBJ, SWING.OBJ
```

3.3.2 LINK Command Qualifiers

The LINK command qualifiers can be used to modify the linker's output, as well as to invoke the debugging and traceback facilities. Linker output consists of an image file and an optional map file. Image file qualifiers, map file qualifiers, and debugging and traceback qualifiers are described in this section.

Table 3–9 summarizes the qualifiers available with the LINK command.

Table 3–9: LINK Command Qualifiers

Qualifier	Action	Default
/[NO]EXECUTABLE	Produces an executable image. /NOEXECUTABLE suppresses production of an image file.	/EXECUTABLE= object-file.EXE
/[NO]SHAREABLE	Creates a shareable image; /NOSHAREABLE generates an executable image.	/NOSHAREABLE
/BRIEF	Produces a summary of the image's characteristics and a list of contributing modules.	Not applicable
/[NO]CROSS_ REFERENCE	Produces cross-reference information for global symbols; /NOCROSS_REFERENCE suppresses cross-reference information.	/NOCROSS_ REFERENCE
/FULL	Produces a summary of the image's characteristics, a list of contributing modules, listings of global symbols by name and by value, and a summary of characteristics of image sections in the linked image.	Not applicable

Table 3–9 (Cont.): LINK Command Qualifiers

Qualifier	Action	Default
/[NO]MAP	Generates a map file; /NOMAP suppresses the map.	/NOMAP (interactive) /MAP=object-file.MAP (batch)
/[NO]DEBUG	Includes the VAX Symbolic Debugger in the executable image and generates a symbol table; /NODEBUG prevents debugger control of the program.	/NODEBUG
/[NO]TRACEBACK	Generates symbolic traceback information when error messages are produced; /NOTRACEBACK suppresses traceback information.	/TRACEBACK

3.3.3 Image-File Qualifiers

This section describes how the LINK command qualifiers affect the production of an image file.

3.3.3.1 /EXECUTABLE Qualifier

The /EXECUTABLE qualifier produces an executable image.

A file specification can follow /EXECUTABLE to designate a name for the image file. For example:

```
$ LINK/EXECUTABLE=TEST CIRCLE
```

This command causes the file CIRCLE.OBJ to be linked, and the executable image generated by the linker to be named TEST.EXE.

The /NOEXECUTABLE qualifier, which suppresses production of the image file, is useful when you want to verify the results of linking an object file before the image is produced. For example:

```
$ LINK/NOEXECUTABLE CIRCLE
```

This command causes the file CIRCLE.OBJ to be linked, but no image file is generated.

By default, /EXECUTABLE is enabled.

3.3.3.2 /SHAREABLE Qualifier

The /SHAREABLE qualifier creates a shareable image. A shareable image is an image that has all of its internal references resolved, but that must be linked with one or more object modules to produce an executable image. For example, a shareable image can contain a library of routines or can be used by the system manager to create a global section for all users.

To include a shareable image as input to the linker, you can insert the shareable image into a shareable-image library and specify the library as input to the LINK command. By default, the linker automatically searches the system-supplied shareable-image library SYS$LIBRARY:IMAGELIB.OLB after searching any libraries you specify on the LINK command line. You can also include a shareable image by using a linker options file. See the *VAX/VMS Linker Reference Manual* for details.

By default, /NOSHAREABLE is enabled. With /NOSHAREABLE, the image produced cannot be linked with other images.

3.3.4 Map File Qualifiers

The map file qualifiers control the generation of a map file and its contents. The /MAP qualifier produces a map file, which you can name by including a file specification. For example:

```
$ LINK/MAP=TEST CIRCLE
```

This command causes the file CIRCLE.OBJ to be linked, and the map file generated by the linker to be named TEST.MAP.

The map file is stored on the default device in the default directory. If you do not include a file specification with /MAP, the map file is given the name of the first input file and a file type of MAP.

With the /MAP qualifier, you can use the qualifiers /BRIEF, /FULL, and /CROSS_REFERENCE. These qualifiers define the type of information included in the map file as follows:

- /BRIEF produces a summary of the image's characteristics and a list of the contributing modules.

- /FULL produces the effects of /BRIEF, plus listings of global symbols by name and by value, and a summary of the characteristics of image sections in the linked image.
- /CROSS_REFERENCE requests cross-reference information, which indicates the object modules that define or refer to global symbols encountered during linking.

In interactive mode, the default is /NOMAP; in batch mode, the default is /MAP. /NOCROSS_REFERENCE is enabled in either mode.

/CROSS_REFERENCE and /FULL can be used to modify the same /MAP qualifier. If you specify neither /BRIEF nor /FULL, the map file by default 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, see the *VAX/VMS Linker Reference Manual.*

3.3.5 Debugging and Traceback Qualifiers

The /DEBUG qualifier indicates that the VAX/VMS Debugger is to be included in the executable image and that a symbol table is to be generated. If you specify LINK/DEBUG, the program will link and execute under the control of the debugger, unless you specify RUN/NODEBUG. At link time, the default is /NODEBUG.

The /TRACEBACK qualifier causes the generation of error messages to be accompanied by symbolic traceback information. This information shows the sequence of calls that transferred control to the program in which the error occurred. /NOTRACEBACK suppresses production of traceback information. The default is /TRACEBACK.

The traceback capability is automatically included with the /DEBUG qualifier; therefore, if you specify both /DEBUG and /NOTRACEBACK, /NOTRACEBACK has no effect.

Chapter 4 contains more information on the VAX/VMS Debugger.

3.3.6 File Qualifiers

The file qualifiers /INCLUDE, /LIBRARY, and /OPTIONS are some of the qualifiers you can apply to input file specifications with the LINK command. Input files can be object files, shareable files previously linked, or library files.

3.3.7 /INCLUDE Qualifier

The /INCLUDE qualifier specifies that the input file is an object module or a shareable image library, and that the modules named are the only ones in the library to be explicitly included as input. In the case of shareable image libraries, the module is the shareable image name. You must specify at least one module name with the /INCLUDE qualifier. The default file type is OLB.

To specify more than one module, use the following format:

```
/INCLUDE={module-name,...}
```

The following example shows the use of the /INCLUDE qualifier with a library named COURSES that contains many modules, among them HISTORY, ALGEBRA, and PHILOSOPHY.

```
$ LINK SCHEDULE,COURSES/INCLUDE=(HISTORY,ALGEBRA,PHILOSOPHY)
```

This command causes the linker to extract the modules HISTORY, ALGEBRA, and PHILOSOPHY from the library COURSES and include them in the executable image named SCHEDULE.

3.3.8 /LIBRARY Qualifier

The /LIBRARY qualifier specifies that the input file is an object-module or shareable-image library, which the linker must search to resolve undefined symbols to which other input modules refer. The default file type for the input file is OLB.

You can use the /LIBRARY qualifier with the /INCLUDE qualifier to modify the same input file specification. In that case, the same library is searched for unresolved references.

The following example shows the addition of the /LIBRARY qualifier to the example given in Section 3.3.7:

```
$ LINK SCHEDULE,COURSES/LIBRARY/INCLUDE=(HISTORY,ALGEBRA,PHILOSOPHY)
```

This example also causes the linker to include the modules HISTORY, ALGEBRA, and PHILOSOPHY in the image file SCHEDULE. However, the /LIBRARY qualifier causes the linker to search the rest of the library COURSES and link in any other modules needed to resolve strong symbolic references in SCHEDULE, HISTORY, ALGEBRA, and PHILOSOPHY.

3.3.9 /OPTIONS Qualifier

The /OPTIONS qualifier specifies that the input file is a linker options file, which can contain input file specifications as well as special instructions recognized only by the linker. See the *VAX/VMS Linker Reference Manual* for a more detailed explanation on the linker options file.

3.3.10 Identification Checking

The VAX PASCAL compiler provides identification information about object modules so that the VAX/VMS Linker can check for consistency among object modules that are linked together and that inherit the same environment files. The compiler records the date and time that an environment file was created and passes this information to the linker in the form of an Entity Ident Consistency Check subrecord.

The linker checks the Entity Ident Consistency Check subrecord of every object module before it links them together. Two object modules that inherit the same environment file must inherit versions of the file with the same identification information. If the creation dates and times are not identical, the linker issues a warning message.

For more information about the Entity Ident Consistency Check subrecords, see the *VAX/VMS Linker Reference Manual.*

3.3.11 Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify input file specifications for the object modules.

  If no file type is specified, the linker assumes that an input file is an object file with the file type OBJ.
- Specify one or more object module library files.

  You can either specify the name of an object module library with the /LIBRARY qualifier, or specify the names of object modules contained in an object module library with the /INCLUDE qualifier. The uses of object module libraries are described in Section 3.3.13.
- Specify an options file.

An options file can contain additional file specifications for the LINK command as well as special linker options. You must use the /OPTIONS qualifier to specify an options file. For more information on options files see the *VAX/VMS Linker Reference Manual*

The linker uses the following default file types for input files:

File Type	File
OBJ	Object module
OLB	Library
OPT	Options file

3.3.12 Linker Output Files

When you enter the LINK command interactively and do not specify any qualifiers, the linker creates only an executable image file. By default, the resulting image file has the same file name as the first object module specified, and a file type of EXE.

In a batch job, the linker creates both an executable image file and storage map file by default. The default file type for map files is MAP.

To specify an alternative name for a map file or image file or to specify an alternative output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. For example:

```
$ LINK UPDATE/MAP=TEST
```

This command generates the files UPDATE.EXE (by default) and TEST.MAP.

```
$ LINK UPDATE/EXE=[PROJECT.EXE]/MAP=[PROJECT.MAP]
```

This command produces the files [PROJECT.EXE]UPDATE.EXE and [PROJECT.MAP]UPDATE.MAP.

3.3.13 Object Module Libraries

In a large development effort, programmers often store the object modules for their subprograms in an object module library. By using an object module library, you can make program modules contained in the library available to other programmers. To link modules contained in a object module library, use the /INCLUDE qualifier and specify the specific modules you want to link. For example:

```
$ LINK GARDEN, VEGETABLES/INCLUDE=(EGGPLANT,TOMATO,BROCCOLI,ONION)
```

This example directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN.

Besides program modules, an object module library can also contain a symbol table with the names of each global symbol in the library, and the name of the module in which they are defined. You specify the name of the object module library containing symbol definitions with the /LIBRARY qualifier. When you use the /LIBRARY qualifier during a link operation, the linker searches the specified library for all unresolved references found in the included modules during compilation.

In the following example, the linker uses the library RACQUETS to resolve undefined symbols in BADMINTON, TENNIS, and RACQUETBALL.

```
$ LINK BADMINTON, TENNIS, RACQUETBALL, RACQUETS/LIBRARY
```

You can define an object module library to be your default library by using the DCL command DEFINE. The linker searches default user libraries for unresolved references after it searches modules and libraries specified in the LINK command. See the *VAX/VMS DCL Dictionary* for more information about the DEFINE command.

For more information about object module libraries, see the *VAX/VMS Linker Reference Manual.*

3.3.14 Linker Error Messages

If the linker detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any error or fatal error conditions occur, that is, errors with severities of E or F, the linker does not produce an image file.

The messages produced by the linker are descriptive, and you do not usually need additional information to determine the specific error. Some common errors that occur during linking are as follows:

- An object module has compilation errors.

  This error occurs when you attempt to link a module that has warnings or errors during compilation. You can usually link compiled modules for which the compiler generated messages, but you should verify that the modules will actually produce the output you expect.

- The input file has a file type other than OBJ and no file type was specified on the command line.

  If you do not specify a file type, the linker assumes the file has a file type of OBJ by default. If the file is not an object file and you do not identify it with the appropriate file type, the linker signals an error message and does not produce an image file.

- You tried to link a nonexistent module.

  The linker signals an error message if you misspell a module name on the command line or if the compilation contains fatal diagnostics.

- A reference to a symbol name remains unresolved.

  An error occurs when you omit required module or library names from the command line and the linker cannot locate the definition for a specified global symbol reference. For example, a main program module OCEAN.OBJ calls the subprograms REEF.OBJ, SHELLS.OBJ, and SEAWEED.OBJ. However, the following LINK command does not reference SEAWEED.OBJ:

  ```
  $ LINK OCEAN, REEF, SHELLS
  ```

  This example produces the following error messages:

  ```
  %LINK-W-NUDFSYMS, 1 undefined symbol
  %LINK-I-UDFSYMS,        SEAWEED
  %LINK-W-USEUNDEF, module "OCEAN" references undefined symbol "SEAWEED"
  %LINK-W-DIAGISUED, completed but with diagnostics
  ```

If an error occurs when you link modules, you can often correct the error by reentering the command string and specifying the correct modules or libraries. If an error indicates that a module cannot be located, you may be linking the program with the wrong VAX PASCAL run-time library.

See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a complete list of linker messages.

3.4 Running a VAX PASCAL Program

Once you have linked your program, you can use the DCL command RUN to execute it. The RUN command has the following form:

```
RUN [/[NO]DEBUG] file-spec
```

/[NO]DEBUG
The /[NO]DEBUG qualifier is optional. Specify the /DEBUG qualifier to request the debugger if the image was not linked with it. You cannot use /DEBUG on images linked with the /NOTRACEBACK qualifier. If the image was linked with the /DEBUG qualifier and you do not want the debugger to prompt, use the /NODEBUG qualifier. The default action depends on whether the file was linked with the /DEBUG qualifier.

file-spec
The name of the executable image you want to be run.

The following example executes the image NOBUGS.EXE without invoking the debugger:

```
$ RUN NOBUGS/NODEBUG
```

See Chapter 4 for more information on debugging programs.

During execution, an image can generate a fatal error called an exception condition. When an exception condition occurs, the system displays an error message. Run-time errors can also be issued by other facilities, such as the VAX/VMS Sort Utility or the VAX/VMS operating system.

VAX PASCAL run-time system error messages have the following form:

```
%PAS-s-code,text
```

s
Severity of the error.

code
An abbreviation of the message text.

text
Explanation of the error.

For example, the following VAX PASCAL run-time error message is issued when you have indexed an array element that is outside the bounds of the array.

```
%PAS-F-ARRINDVAL, array index value is out of range
```

For a complete list of VAX PASCAL run-time error messages, see Appendix A.

Chapter 4

Using the VAX/VMS Debugger

This chapter is an introduction to using the VAX/VMS Debugger with VAX PASCAL programs. Included in the chapter are the following:

- An overview of the debugger
- Enough information to get you started using the debugger
- A sample terminal session that demonstrates using the debugger to find a bug in a VAX PASCAL program
- A list of the debugger commands by function

For complete reference information on the VAX/VMS Debugger, see the *VAX/VMS Debugger Reference Manual.* Online HELP is available during debugging sessions.

4.1 Overview of the Debugger

A debugger is a tool to help you locate run-time errors quickly. It is used with a program that has already been compiled and linked successfully, with no errors reported, but that does not run correctly. For example, the output may be obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively, step by step, until you locate the point at which the program stopped working correctly.

The VAX/VMS Debugger is a *symbolic* debugger, which means that you can refer to program locations by the symbols (names) you used for those locations in your program—the names of variables, routines, labels, and so on. You do not have to use virtual addresses to refer to memory locations.

The debugger recognizes the syntax, expressions, data typing, and other constructs of VAX PASCAL, as well as other VAX/VMS supported languages.

Therefore, if your program is written in more than one language, you can change from one language to another in the course of a debugging session. The current source language determines the format used for entering and displaying data, as well as other features that have language-specific settings (for example, comment characters, operators and operator precedence, and case sensitivity or insensitivity).

By issuing debugger commands at your terminal, you can do the following:

- Start, stop, and resume the program's execution
- Trace the execution path of the program
- Monitor selected locations, variables, or events
- Examine and modify the contents of variables, or force events to occur
- Test the effect of some program modifications without having to edit, recompile, and relink the program

Such techniques can enable you to isolate an error in your code much more quickly than you could without the debugger. Once you have found the error in the program, you can then edit the source code and compile, link, and run the corrected version.

4.2 Features of the Debugger

The VAX/VMS Debugger provides various features that help you to debug your programs:

- Online HELP. Online HELP is always available during a debugging session. Online HELP contains information on all of the debugger commands and also for selected topics.
- Source code display. The debugger lets you display lines of source code during a debugging session.
- Screen mode. In screen mode, you can display and capture various kinds of information in scrollable windows that can be moved around the screen and resized. Source, instruction, and register displays are automatically updated. You can selectively direct debugger input, output, and diagnostic messages to displays. (Screen mode is best

displayed on VT100-series or VT200-series terminals or MicroVAX workstations.)

- Keypad mode. When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad (if you have a VT100, VT52, or LK201 keypad). Using the keypad can be more efficient than typing commands because the keypad keys can save keystrokes. (See Figure 4–1 for a diagram of the keypad key functions.)
- Source editing. As you find errors during a debugging session, you can use the EDIT command to invoke any editor available on your system. You specify the editor you want with the SET EDITOR command.
- Command procedures. You can direct the debugger to execute a command procedure (a file of debugger commands) to recreate a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session.
- Symbol definitions. You can define your own symbols to represent lengthy commands, address expressions, or values in abbreviated form.
- Initialization files. You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. In addition, you may want to have special initialization files for debugging specific programs. When you invoke the debugger, those commands will be executed automatically.
- Log files. You can record the commands you enter during a debugging session and the debugger's responses to those commands in a log file. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

4.3 Getting Started with the Debugger

This section explains how to use the debugger and provides VAX PASCAL examples. The intent is to get you started using the debugger; therefore, only basic functions are covered. For more detailed information, see the *VAX/VMS Debugger Reference Manual*. Remember that online HELP is immediately available to you during a debugging session when you type the HELP command at the DBG> prompt.

4.3.1 Compiling and Linking to Prepare for Debugging

The following example shows how to compile and link a VAX PASCAL program (consisting of a single compilation unit named INVENTORY) so that subsequently you will be able to use the debugger.

```
$ PASCAL/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

The /DEBUG qualifier on the PASCAL command causes the compiler to write the debug symbol records associated with INVENTORY into the object module, INVENTORY.OBJ. These records allow you to use the names of variables and other symbols declared in INVENTORY with debugger commands. (If your program has several compilation units, you must compile each unit that you want to debug with the /DEBUG qualifier).

You should use the /NOOPTIMIZE qualifier when you compile in preparation for debugging. Without this qualifier, the resulting object code is optimized, possibly causing the contents of some program locations to be inconsistent with what you might expect from the source code. (After the program has been debugged, you will probably want to recompile it without the /NOOPTIMIZE qualifier, because optimization may reduce a program's size and increase the execution speed.)

The /DEBUG qualifier on the LINK command causes the linker to include all symbol information that is contained in INVENTORY.OBJ in the executable image. The qualifier also causes the VAX/VMS image activator to start the debugger at run time. (If your program has several object modules, you may need to specify other modules in the LINK command).

4.3.2 Starting and Terminating a Debugging Session

To invoke the debugger, issue the DCL command RUN. The following message will appear on your screen.

```
$ RUN INVENTORY

                    VAX DEBUG Version 4.n

%DEBUG-I-INITIAL, language is PASCAL, module set to 'INVENTORY'
DBG>
```

The DBG> prompt indicates that you can now type debugger commands. At this point, if you type the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

To interrupt a debugging session while a debugging command is in progress, press CTRL/Y to return to DCL level. For example, if your program loops or otherwise fails to complete execution, press CTRL/Y to stop the debugging session and return to DCL level.

After you have used CTRL/Y to interrupt a debugging session, you can resume the session by using the CONTINUE or DEBUG command at DCL level. In general, you use the CONTINUE command at DCL level to return to where you interrupted the debugging session. If you interrupted the session with CTRL/Y because of an infinite loop, you use the DCL command DEBUG instead of the CONTINUE command. The DEBUG command returns you to the debugger prompt so that you can type another command. For example:

```
DBG> GO
       .
       .
       .
CTRL/Y
Interrupt

$ DEBUG
DBG>
```

The following message indicates that your program has completed successfully:

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

To end a debugging session, type the EXIT command at the DBG> prompt or press CTRL/Z:

```
DBG> EXIT
$
```

4.3.3 Issuing Debugger Commands

You can issue debugger commands any time you see the debugger prompt (DBG>). To enter a command, type it at the keyboard and press the RETURN key. You can issue several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the line with a hyphen (-).

Alternatively, you can use the numeric keypad. In addition to the STEP, GO, SHOW CALLS, and EXAMINE commands, several functions that manipulate screen-mode displays are bound to the keys. Figure 4–1 identifies the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

Most keypad keys have three predefined functions—DEFAULT, GOLD, and BLUE. (The PF1 key is commonly known as the GOLD key, and the PF4 key is commonly known as the BLUE key.) To obtain a key's DEFAULT function, press the key. To obtain its GOLD function, first press the PF1 (GOLD) key, and then the key. To obtain its BLUE function, first press the PF4 (BLUE) key, and then the key.

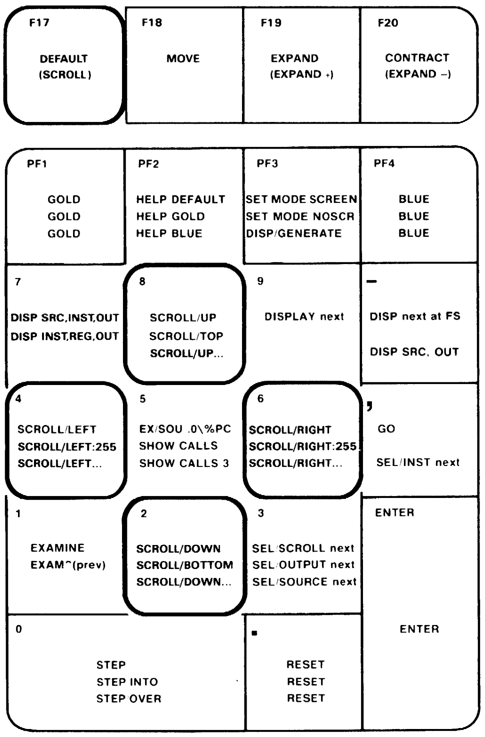
In Figure 4–1, the DEFAULT, GOLD, and BLUE functions are listed within each key's outline, from top to bottom respectively. For example, pressing keypad key 0 issues the STEP command (DEFAULT function); pressing key PF1 and then key 0 issues the STEP/INTO command (GOLD function); pressing key PF4 and then key 0 issues the STEP/OVER command (BLUE function).

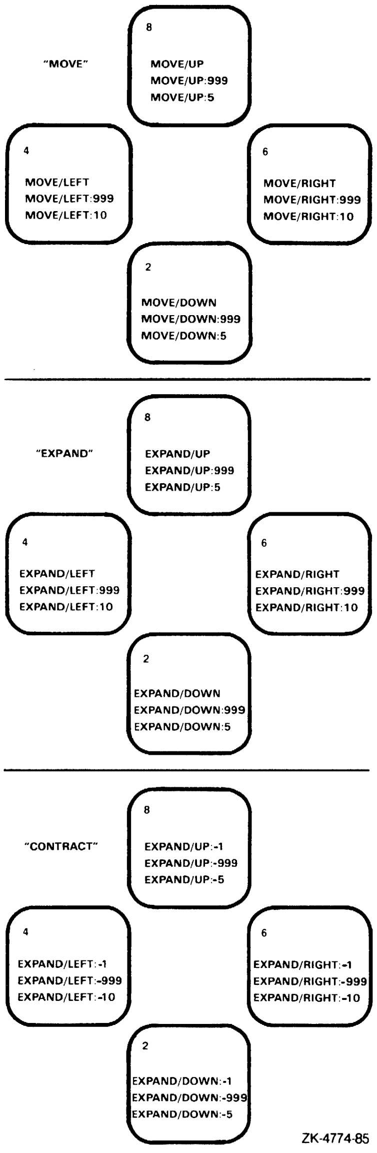
Type the HELP KEYPAD command to get help on the keypad key definitions.

4.3.4 Viewing Your Source Code

The debugger provides two methods for viewing source code: noscreen mode and screen mode. By default, when you invoke the debugger, you are in noscreen mode, but you may find that it is easier to view your source code with screen mode. Both modes are briefly described in the following sections.

Figure 4–1: Keypad Key Functions Predefined by the Debugger





LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT-100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

4.3.4.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To invoke noscreen mode from screen mode, type SET MODE NOSCREEN. See the sample debugging session in Section 4.4 for a demonstration of noscreen mode.

In noscreen mode, you can use the TYPE command to display one or more source lines. For example, the following command displays line 3 of the module that is currently executing:

```
DBG> TYPE 3
3:     I := 3;
DBG>
```

The display of source lines is independent of program execution. To display source code from a module other than the one currently executing, use the TYPE command with a path name to specify the module. For example, the following command displays lines 1 through 8 of module TEST:

```
DBG> TYPE TEST\1:8
```

4.3.4.2 Screen Mode

To invoke screen mode, press keypad key PF3. In screen mode, by default the debugger splits the screen into three displays named SRC, OUT, and PROMPT; the following example shows how your screen will appear in screen mode.

```
- SRC: module SUM -scroll-source-----------------------
     1: PROGRAM SUM (INPUT,OUTPUT);
     2:
     3: VAR
     4:   I, Highest, Total:INTEGER;
     5:
     6: BEGIN
     7: Total := 0;
->   8: WRITE('Enter an integer; type 0 to quit:');
     9: READ (highest);
    10: WHILE Highest <> 0 DO
    11:          BEGIN
- OUT -output------------------------------------------
stepped to SUM\%LINE 8
SUM\Total: 0


- PROMPT -error-program-prompt-------------------------
```

```
DBG> STEP 2
DBG> EXAMINE Total
DBG>
```

The SRC display, at the top of the screen, shows the source code of the program that is currently executing. An arrow in the left column points to the next line to be executed, which corresponds to the current location of the program counter (PC). The line numbers, which are assigned by the compiler, match those in a listing file.

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output.

In the example, the two debugger commands that have been issued (STEP 2 and EXAMINE Total) are displayed.

The OUT display, in the center of the screen, captures the debugger's output in response to the commands that you issue.

The SRC and OUT displays can be scrolled to display information beyond the window's edge. Press keypad key 8 to scroll up and keypad key 2 to scroll down. Use keypad key 3 to change the display to be scrolled (by default, the SRC display is scrolled). Scrolling a display does not affect program execution.

If the debugger cannot locate source lines for the currently executing module, it tries to display source lines in the next module down on the call stack for which source lines are available and issues the following message:

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
        Displaying source in a caller of the current routine.
```

Source lines may not be available for the following reasons:

- The PC is within a system routine, or a shareable image routine for which no source code is available.
- The PC is within a routine that was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object files).

4.3.5 Controlling and Monitoring Program Execution

This section discusses the following topics:

- Starting and resuming program execution with the GO command
- Stepping through the program's code with the STEP command
- Determining the current location of the program counter (PC) with the SHOW CALLS command
- Suspending program execution with breakpoints
- Tracing program execution with tracepoints
- Monitoring changes in variables with watchpoints

4.3.5.1 Starting and Resuming Program Execution

There are two commands for starting or resuming program execution: GO and STEP. The GO command starts execution. The STEP command lets you execute a specified number of source lines or instructions.

The GO Command

The GO command starts program execution, which continues until forced to stop. The GO command is used most often in conjunction with breakpoints, tracepoints, and watchpoints. If you set a breakpoint in the path of execution and then issue the GO command, execution will be suspended when the program reaches that breakpoint. If you set a tracepoint, the path of execution through that tracepoint will be monitored. If you set a watchpoint, execution will be suspended when the value of the watched variable changes.

You can also use the GO command to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger will take over and display the DBG> prompt so that you can issue commands. If you are using screen mode, the pointer in the source display will indicate where execution stopped. You can use the SHOW CALLS command (see Section 4.3.5.2) to identify the currently active routine calls (the call stack).

In the case of an infinite loop, the program will not terminate, so the debugger prompt will not reappear. To obtain the prompt, interrupt the program by pressing CTRL/Y and then issue the DCL command DEBUG. You can then look at the source display and a SHOW CALLS display to locate the PC.

The STEP Command

The STEP command (which you can use either by typing STEP or by pressing the keypad 0 key) allows you to execute a specified number of source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next CALL instruction.

By default, the STEP command executes a single source line at a time. In the following example, the STEP command executes one line, reports the action ("stepped to . . . "), and displays the line number (27) and source code of the next line to be executed:

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
     27:    X := X + 1;
DBG>
```

The PC is now at the first machine code instruction for line 27 of the module TEST; line 27 is in COUNT, a routine within the module TEST. "TEST\COUNT\%LINE 27" is a *path name*. The debugger uses path names to refer to symbols. (You do not need to use a path name in referring to a symbol, however, unless the symbol is not unique. If the symbol is not unique, the debugger will issue an error message. See Section 4.3.7.2 for more information on resolving multiply defined symbols.)

The STEP command can execute a number of lines at a time. In the following example, the STEP command executes three lines:

```
DBG> STEP 3
```

Note that only those source lines for which code instructions were generated by the compiler are recognized as executable lines by the debugger. The debugger skips over any other lines—for example, comment lines.

Also, if a line has more than one statement on it, the debugger will execute all the statements on that line as part of the single step.

You can specify different stepping modes, such as stepping by instruction rather than by line (SET STEP INSTRUCTION). Also, by default, the debugger steps over called routines—execution is not suspended within a called routine, although the routine is executed. By issuing the SET STEP INTO command, you tell the debugger to suspend execution within called routines, as well as within the currently executing module.

4.3.5.2 Determining the Current Location of the Program Counter

The SHOW CALLS command lets you determine the current location of the program counter (PC) (for example, after returning to the debugger following a CTRL/Y interrupt). The command shows a traceback that lists the sequence of calls leading to the currently executing routine. For example:

```
DBG> SHOW CALLS
 module name        routine name        line      rel PC    abs PC

*TEST               PRODUCT               18    00000009  0000063C
*TEST               COUNT                 47    00000009  00000647
*MY_PROG            MY_PROG               21    0000000D  00000653
DBG>
```

For each routine (beginning with the currently executing routine), the debugger displays the following information:

- The name of the module that contains the routine
- The name of the routine
- The line number at which the call was made (or at which execution is suspended, in the case of the current routine)
- The corresponding PC addresses (the relative PC address from the start of the routine, and the absolute PC address of the program)

This example indicates that execution is currently at line 18 of routine PRODUCT (in module TEST), which was called from line 47 of routine COUNT (in module TEST), which was called from line 21 of routine MY_PROG (in module MY_PROG).

4.3.5.3 Suspending Program Execution

The SET BREAK command lets you select breakpoints, which are locations at which the program will stop running. When you reach a breakpoint, you can issue commands to check the call stack, examine the current values of variables, and so on.

The following example shows a typical use of the SET BREAK command.

```
DBG> SET BREAK COUNT
DBG> GO
    .
    .
    .
break at PROG2\COUNT
     54:  PROCEDURE COUNT(X,Y:INTEGER);
DBG>
```

In this example, the SET BREAK command sets a breakpoint on the procedure COUNT. The GO command then starts execution. When the procedure COUNT is encountered, execution is suspended. The debugger announces that the breakpoint at COUNT has been reached ("break at . . . "), displays the source line (54) where execution is suspended, and prompts you for another command. At this breakpoint, you could step through the procedure COUNT, using the STEP command, and use the EXAMINE command (see Section 4.3.6.1) to check on the current values of X and Y.

When using the SET BREAK command, you can specify program locations using various kinds of *address expressions* (for example, line numbers, routine names, instructions, virtual memory addresses, and byte offsets). With high-level languages, you typically use routine names, labels, or line numbers, possibly with path names to ensure uniqueness.

Routine names and labels should be specified as they appear in the source code. Line numbers may be derived from either a source code display or a listing file. When specifying a line number, use the prefix %LINE. (Otherwise, the debugger will interpret the line number as a memory location.) For example, the next command sets a breakpoint at line 41 of the currently executing module; the debugger will suspend execution when the PC is at the start of line 41.

```
DBG> SET BREAK %LINE 41
```

Note that you can set breakpoints only on lines that resulted in machine code instructions. The debugger warns you if you try to do otherwise (for example, on a comment line). If you want to pick a line number in a module other than the one currently executing, you need to specify the module's name in a path name. For example:

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

You do not always have to specify a particular program location, such as line 58 or COUNT, to set a breakpoint. You can set breakpoints on events, such as exceptions. You can use the SET BREAK command with a qualifier, but no parameter, to break on every line, or on every CALL instruction, and so on. For example:

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

You can conditionalize a breakpoint (with a WHEN clause) or specify that a list of commands be executed at the breakpoint (with a DO clause on the debugger command). For example, the next command sets a breakpoint on the label LOOP3. The DO (EXAMINE TEMP) clause causes the value of the variable TEMP to be displayed whenever the breakpoint is triggered.

```
DBG> SET BREAK LOOP3 DO (EXAMINE TEMP)
DBG> GO
     .
     .
     .
break at COUNT\LOOP3
     37:    LOOP3: FOR I := 1 TO 10 DO
COUNT\TEMP:   284.19
DBG>
```

To display the currently active breakpoints, issue the SHOW BREAK command:

```
DBG> SHOW BREAK
breakpoint at SCREEN_IO\%LINE 58
breakpoint at COUNT\LOOP3
   do (EXAMINE TEMP)
     .
     .
     .
DBG>
```

To cancel a breakpoint, issue the CANCEL BREAK command, specifying the program location exactly as you did when setting the breakpoint. The CANCEL BREAK/ALL command cancels all breakpoints.

4.3.5.4 Tracing Program Execution

The SET TRACE command lets you select tracepoints, which are locations for tracing the execution of your program without stopping its execution. After setting a tracepoint, you can start execution with the GO command and then monitor the PC's path, checking for unexpected behavior. By setting a tracepoint on a routine, you can also monitor the number of times the routine is called.

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. It can also display other information that you have specified (for example, the value of a specified variable, as illustrated later in this section). However, at tracepoints, the program continues executing, and the debugger prompt is not displayed. For example:

```
DBG> SET TRACE COUNT
DBG> GO
    .
    .
    .
trace at PROG2\COUNT
     54:  PROCEDURE COUNT(X,Y:INTEGER);
    .
    .
    .
```

When using the SET TRACE command, you specify address expressions, qualifiers, and optional clauses exactly as with the SET BREAK command.

The /LINE qualifier causes the SET TRACE command to trace every line and is a convenient means of checking the execution path. By default, lines are traced within all called routines, as well as the currently executing routine. If you do not want to trace system routines or routines in shareable images, use the /NOSYSTEM or /NOSHARE qualifiers. For example:

```
DBG> SET TRACE/LINE/NOSYSTEM/NOSHARE
```

The /SILENT qualifier suppresses the trace message and source code display. This is useful when you want to use the SET TRACE command to execute a debugger command at the tracepoint. For example:

```
DBG> SET TRACE/SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
    .
    .
    .
SCREEN_IO\CLEAR\STATUS:    99
    .
    .
    .
```

4.3.5.5 Monitoring Changes in Variables

The SET WATCH command lets you set watchpoints that will be monitored continuously as your program executes. Watchpoints can be program variables or arbitrary program locations. If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

To set a watchpoint on a variable, specify the variable's name with the SET WATCH command. For example, the following command sets a watchpoint on the variable TOTAL:

```
DBG> SET WATCH TOTAL
```

Subsequently, every time the program modifies the value of TOTAL, the watchpoint is triggered.

The following example shows the effect on program execution when your program modifies the contents of a watched variable.

```
DBG> SET WATCH TOTAL
DBG> GO
     .
     .
     .
watch of SCREEN_IO\TOTAL\%LINE 13
     13:   TOTAL := TOTAL + 1;
    old value: 16
    new value: 17
break at SCREEN_IO.%LINE 14
     14:   POP(TOTAL);
DBG>
```

In this example, a watchpoint is set on the variable TOTAL, and the GO command is issued to start execution. When the value of TOTAL changes, execution is suspended. The debugger announces the event ("watch of . . . "), identifying where TOTAL changed (line 13) and the associated source line. The debugger then displays the old and new values and announces that execution has been suspended at the start of the next line (14). (The debugger reports "break at . . . ", but this is not a breakpoint; it is the effect of the watchpoint.) Finally, the debugger prompts for another command.

When a change in a variable occurs at a point other than the start of a source line, the debugger gives the line number plus the byte offset from the start of the line.

4.3.6 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables, and evaluate expressions. It also notes restrictions on the use of these commands in VAX PASCAL.

4.3.6.1 Displaying the Values of Variables

To display the current value of a variable, use the EXAMINE command. The EXAMINE command has the following form:

```
EXAMINE variable-name
```

The debugger recognizes the compiler-generated data type of the specified variable and retrieves and formats the data accordingly. The following examples show some uses of the EXAMINE command.

Example 1

Examine a varying string variable:

```
DBG> EXAMINE VARY_STRING
DEBUG_TEST\VARY_STRING:     '12345'
DBG>
```

Example 2

Examine three integer variables:

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

Example 3

Examine a two-dimensional array of integers (three per dimension):

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
   [1,1]:        27
   [1,2]:        31
   [1,3]:        12
   [2,1]:        15
   [2,2]:        22
   [2,3]:        18
DBG>
```

Example 4

Examine element 4 of a one-dimensional array of characters:

```
DBG> EXAMINE CHAR_ARRAY[4]
PROG2\CHAR_ARRAY[4]: 'm'
DBG>
```

The EXAMINE command can be used with any kind of address expression, not just a variable name, to display the contents of a program location. The debugger associates certain default data types with untyped locations. You can override the defaults for typed and untyped locations if you want the data to be interpreted and displayed in some other data format.

See Section 4.3.6.3 for an explanation of differences between the EXAMINE and EVALUATE commands.

4.3.6.2 Changing the Values of Variables

To change the value of a variable, use the DEPOSIT command. The DEPOSIT command has the following form:

```
   DEPOSIT variable-name [:]= value
```

The DEPOSIT command is like an assignment statement in VAX PASCAL.

In the following examples, the DEPOSIT command assigns new values to different variables. The debugger checks that the value assigned, which can be a language expression, is consistent with the data type and dimensional constraints of the variable.

Example 1

Deposit a string value (it must be enclosed in quotation marks or apostrophes):

```
DBG> DEPOSIT PARTNUMBER := "WG-7619.3-84"
```

Example 2

Deposit an integer expression:

```
DBG> DEPOSIT WIDTH := CURRENTWIDTH + 10
```

Example 3

Deposit element 12 of an array of characters (you cannot deposit an entire array aggregate with a single DEPOSIT command, only an element):

```
DBG> DEPOSIT C_ARRAY[12] = 'K'
```

As with the EXAMINE command, the DEPOSIT command lets you specify any kind of address expression, not just a variable name. You can override the defaults for typed and untyped locations if you want the data to be interpreted in some other data format.

4.3.6.3 Evaluating Expressions

To evaluate a language expression, use the EVALUATE command. The EVALUATE command has the following form:

```
   EVALUATE language-expression
```

The debugger recognizes the operators and expression syntax of the currently set language. In the following example, the value 45 is assigned to the integer variable WIDTH; the EVALUATE command then obtains the sum of the current value of WIDTH plus 7:

```
DBG> DEPOSIT WIDTH := 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

In the next example, the values TRUE and FALSE are assigned to the Boolean variables WILLING and ABLE, respectively; the EVALUATE command then obtains the logical conjunction of these values:

```
DBG> DEPOSIT WILLING := TRUE
DBG> DEPOSIT ABLE := FALSE
DBG> EVALUATE WILLING AND ABLE
False
DBG>
```

The following example shows the way in which the EVALUATE and EXAMINE commands are similar. When the expression following the command is a variable name, the value reported by the debugger is the same for either command:

```
DBG> DEPOSIT WIDTH := 45
DBG> EVALUATE WIDTH
45
DBG> EXAMINE WIDTH
SIZE\WIDTH:   45
```

The following example shows an important difference between the EVALUATE and EXAMINE commands:

```
DBG> EVALUATE WIDTH + 7
52
DBG> EXAMINE WIDTH + 7
SIZE\WIDTH:   131584
```

With the EVALUATE command, WIDTH + 7 is interpreted as a language expression, which evaluates to 45 + 7, or 52. With the EXAMINE command, WIDTH + 7 is interpreted as an address expression: 7 bytes are added to the address of WIDTH, and whatever value is in the resulting address is reported (in this instance, 131584).

4.3.6.4 Notes on VAX PASCAL Support

In general, the debugger supports the data types and operators of VAX PASCAL and of the other debugger-supported languages. However, there are important language-specific limitations. (To get information on the supported data types and operators of any of the languages, type the HELP LANGUAGE command at the DBG> prompt.)

In general, you can examine, evaluate, and deposit into variables, record fields, and array components. An exception to this occurs under the following circumstances: if a variable is not referenced in a program, the VAX PASCAL compiler may not allocate the variable. If the variable is not allocated and you try to examine it or deposit into it, you will receive an error message.

When depositing data into variables, the debugger truncates the high-order bits if the value being deposited is larger than the variable; it fills the high-order bits with zeros if the value being deposited is smaller than the variable. If the deposit violates the rules of assignment compatibility, the debugger displays an informational message.

Automatic variables can be examined and can have values deposited into them; however, since automatic variables are allocated in stack storage and are contained in registers, their values are considered undefined until the variables are initialized or assigned a value. For example:

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

In this example, the value of variable X should be considered undefined until after a value has been assigned to X.

In addition, although you may examine a VARYING OF CHAR string, it is not possible to examine the LENGTH field. For example, the following is not supported:

```
DBG>EXAMINE VARY_STRING.LENGTH
```

Because the current LENGTH of a VARYING string is the first word, you should do the following to examine the LENGTH:

```
DBG>EXAMINE/WORD VARY_STRING
```

It should also be noted that the typecast operator(::) is not permitted when evaluating VAX PASCAL expressions.

4.3.7 Controlling Symbol References

In most cases, the way the debugger handles symbols is transparent to you. However, the following two areas may require action on your part:

- Module setting
- Multiply-defined symbols

4.3.7.1 Module Setting

To facilitate symbol searches, the debugger loads symbol records from the executable image into a run-time symbol table (RST), where they can be accessed efficiently. Unless a symbol record is in the RST, the debugger cannot recognize or properly interpret that symbol.

Because the RST takes up memory, the debugger loads it dynamically, anticipating what symbols you might want to reference in the course of execution. The loading process is called *module setting,* because all of the symbol records of a given module are loaded into the RST at one time.

At debugger startup, only the module containing the image transfer address is set. Each time the debugger interrupts program execution, it sets the module surrounding the current PC location. This lets you reference the symbols that should be visible at that location.

If you try to reference a symbol in a module that has not been set, the debugger will issue an error message. For example:

```
DBG> EXAMINE K
%DEBUG-W-NOSYMBOL, symbol 'K' is not in symbol table
DBG>
```

You must then use the SET MODULE command to set the module containing that symbol manually:

```
DBG> SET MODULE MOD3
DBG> EXAMINE K
MOD3\ROUT2\K: 26
DBG>
```

The SHOW MODULE command lists the modules of your program and identifies which modules have been set.

Note that dynamic module setting may slow the debugger down as more and more modules are set. If performance becomes a problem, you can use the CANCEL MODULE command to reduce the number of set modules, or you can disable dynamic module setting by issuing the SET MODE NODYNAMIC command. (The SET MODE DYNAMIC command enables dynamic module setting.)

4.3.7.2 Resolving Multiply-Defined Symbols

The debugger finds the symbols that you reference in commands according to the scope and visibility rules of the currently set language. In general, the debugger first looks for a symbol within the block or routine surrounding the current PC location. If the symbol is not found in that scope region, the debugger searches the nesting program unit, then its nesting unit, and so on. (The precise manner depends on the currently set language and guarantees that the proper declaration of a multiply-defined symbol is selected.)

The debugger must let you reference symbols throughout your program, not just those that are visible at the current PC location, to allow you to do such things as set breakpoints in arbitrary areas and examine arbitrary variables. Therefore, if the symbol is not visible at the current PC location, the debugger also searches other scope regions. First, it looks within the currently executing routine, then the caller of that routine, then its caller, and so on, until the symbol is found. Symbolically, this search list is denoted 0,1,2, . . . ,*n*, where *n* is the number of calls in the call stack. Within each of these scope regions, the debugger uses the visibility rules of the currently set language to locate symbols.

If the debugger cannot resolve an ambiguity, it issues an error message. For example:

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG>
```

In the above case, you can use a path name prefix to uniquely specify a declaration of the given symbol. First, use the SHOW SYMBOL command to identify all path names associated with the given symbol; then use the desired path name when referencing the symbol. For example:

```
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
DBG> EXAMINE MOD4\ROUT2\Y
MOD4\ROUT2\Y: 12
DBG>
```

If you need to refer to a particular declaration of Y repeatedly, use the SET SCOPE command to establish a new default scope for symbol lookup. Then, references to Y without a path-name prefix will specify the declaration of Y that is visible in the new scope region. For example:

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

To display the current scope for symbol lookup, use the SHOW SCOPE command. To restore the default scope, use the CANCEL SCOPE command.

4.4 Sample Debugging Session

The example program Sum is listed below, with the line numbers assigned by the compiler. This program prompts for a number and prints the sum of integers from one through the number entered. The bug in program Sum is obvious—the value of the variable Total is not reset to 0 when a new number is entered—but the example illustrates some simple debugging commands.

```
1       PROGRAM Sum (INPUT, OUTPUT);
2
3       VAR
4          I, Highest, Total : INTEGER;
5
6       BEGIN
7       Total := 0;
8       WRITE ('Enter an integer; type 0 to quit : ');
9       READ (Highest);
10      WHILE Highest <> 0 DO
11         BEGIN
12         FOR I := 1 TO Highest DO
13            BEGIN
14            Total := Total + I;
15            END;
16         WRITELN ('The sum of integers from 1 to ', Highest:3,
17            ' is ', Total:4);
18         WRITE ('Enter an integer; type 0 to quit : ');
19         READ (Highest);
20         END;
21      END.
```

Initially, you would compile, link, and run the program as follows:

```
$ PASCAL SUM
$ LINK SUM
$ RUN SUM
Enter an integer; type 0 to quit : 5
The sum of integers from 1 to   5 is   15
Enter an integer; type 0 to quit : 4
The sum of integers from 1 to   4 is   25
Enter an integer; type 0 to quit : 0
$
```

The program returns a correct sum for the first number you enter, but the sum for the second number is too high.

```
$ PASCAL/LIST/DEBUG/NOOPTIMIZE SUM
$ LINK/DEBUG SUM
$ PRINT SUM
```

The PRINT command prints the listing, which shows the compiler-generated line numbers. You are now ready to begin a debugging session. The callout numbers in the terminal session below are keyed to the notes that follow.

```
$ RUN SUM          ❶

                 VAX DEBUG Version 4.X

%DEBUG-I-INITIAL, language is PASCAL, module set to 'SUM'
DBG>SET BREAK %LINE 9         ❷
DBG>SET BREAK %LINE 19
DBG>GO         ❸
break at SUM\%LINE 9  ❹
         9:          READ (Highest);
DBG>EXAMINE TOTAL          ❺
SUM\TOTAL: 0
DBG>GO         ❻
Enter an integer; type 0 to quit : 5
The sum of integers from 1 to   5 is   15
break at SUM\%LINE 19
        19:          READ (Highest); ❼
DBG>EXAMINE TOTAL          ❽
SUM\TOTAL: 15
DBG>DEPOSIT TOTAL=0        ❾
DBG>GO         ❿
Enter an integer; type 0 to quit : 4
The sum of integers from 1 to   4 is   10
break at SUM\%LINE 19
        19:          READ (Highest);
DBG>GO
Enter an integer; type 0 to quit : 0       ⓫
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG>EXIT         ⓬
$
```

❶ When you enter the RUN command, the debugger displays an informational message and displays the DBG> prompt.

❷ You decide that the problem may lie with the initialization of Total. You can test this hypothesis by examining the value of Total each time you enter a new number. To do this, you set two breakpoints, one at each READ procedure call that reads a number from the terminal.

❸ The GO command starts the execution of the program. The debugger tells you where in the program execution begins.

❹ When the first READ procedure call is reached, the debugger interrupts the program's execution and prompts you to enter a command.

❺ You examine the variable Total. Its value is zero, as expected at this point.

❻ The GO command continues the execution of the program, which prompts for a number. The program's response to the number you enter is correct.

❼ The debugger reaches the breakpoint at the second READ procedure call.

❽ You examine the variable Total. Its value is 15, not 0 as it should be. The value of Total needs to be reset to 0 before a new number is read.

❾ The DEPOSIT command replaces the contents of Total with 0. This deposit allows the program to return a correct result the next time the loop is executed.

⑩ The GO command continues program execution. The result is correct.

⑪ When you enter a 0 to the prompt, the program exits. The debugger displays a message indicating the termination status.

⑫ The EXIT command terminates the debugging session.

You can now correct the program so that it initializes the variable Total correctly.

4.5 Debugger Command Summary

Table 4–1 lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

During a debugging session, you can get online HELP on any command and its qualifiers by typing the HELP command followed by the name of the command in question, in the following form:

```
HELP command
```

Table 4–1: Debugger Command Summary

Command	Description
	Starting and Terminating a Debugging Session
($) RUN[1]	Invokes the debugger if LINK/DEBUG was used
($) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed
CTRL/Z or EXIT	Ends a debugging session, executing all exit handlers
QUIT	Ends a debugging session without executing any exit handlers declared in the program
CTRL/Y	Interrupts a debugging session, returning you to DCL level
CTRL/C	Has the same effect as CTRL/Y, unless the program has a CTRL/C service routine

[1]This is a DCL command, not a debugger command.

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Starting and Terminating a Debugging Session	
($) CONTINUE[1]	Resumes a debugging session after a CTRL/Y interruption
($) DEBUG[1]	Resumes a debugging session after a CTRL/Y interruption but returns you to the debugger prompt
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH)
SPAWN	Creates a subprocess; lets you issue DCL commands without interrupting your debugging context (similar to the DCL command SPAWN)
Controlling and Monitoring Program Execution	
GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ STEP	Establishes or displays the default qualifiers for the STEP command
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ BREAK	Sets, displays, or cancels breakpoints
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ TRACE	Sets, displays, or cancels tracepoints
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ WATCH	Sets, displays, or cancels watchpoints
$\left\{\begin{array}{l}\text{SET}\\\text{CANCEL}\end{array}\right\}$ EXCEPTION BREAK	Sets or cancels exception breakpoints

[1]This is a DCL command, not a debugger command.

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Controlling and Monitoring Program Execution	
SHOW CALLS	Identifies the currently active routine calls
SHOW STACK	Gives additional information about the currently active routine calls
CALL	Calls a routine
Examining and Manipulating Data	
EXAMINE	Displays the value of a variable or the contents of a program location
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression
Controlling Type Selection and Symbolization	
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\\\text{CANCEL}\end{array}\right\}$ TYPE	Establishes the type to be associated with untyped program locations, displays the type, or restores the type
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Controlling Symbol Lookup	
SHOW SYMBOL	Displays symbols in your program
{ SET / SHOW / CANCEL } MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module
{ SET / SHOW / CANCEL } IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image
SET MODE [NO]DYNAMIC	Controls whether or not modules are set automatically when the debugger interrupts execution
ALLOCATE	Expands the debugger's memory pool to let you set more modules
{ SET / SHOW / CANCEL } SCOPE	Establishes, displays, or restores the scope for symbol lookup
Displaying Source Code	
TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
{ SET / SHOW / CANCEL } SOURCE	Creates, displays, or cancels a source directory search list
SEARCH	Searches the source code for the specified string
{ SET / SHOW } SEARCH	Establishes or displays the default qualifiers for the SEARCH command
{ SET / SHOW } MAX_SOURCE_FILES	Establishes or displays the maximum number of source files that may be kept open at one time
{ SET / SHOW } MARGINS	Establishes or displays the left and right margin settings for displaying source code

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Using Screen Mode	
SET MODE [NO]SCREEN	Enables or disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
{SET, SHOW, CANCEL} DISPLAY	Creates, identifies, or deletes a display
{SET, SHOW, CANCEL} WINDOW	Creates, identifies, or deletes a window definition
SELECT	Selects a display for a display attribute
SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display into another display
EXTRACT	Saves a display or the current screen state into a file
EXPAND	Expands or contracts a display
MOVE	Moves a display across the screen
{SET, SHOW} TERMINAL	Establishes or displays the height and width of the screen
CTRL/W or DISPLAY/REFRESH	Refreshes the screen
Editing Source Code	
EDIT	Invokes an editor during a debugging session
{SET, SHOW} EDITOR	Establishes or identifies the editor invoked by the EDIT command

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Defining Symbols	
DEFINE	Defines a symbol as an address, command, or value
DELETE or UNDEFINE	Deletes symbol definitions
{ SET / SHOW } DEFINE	Establishes or displays the default qualifier for the DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined
Using Keypad Mode	
SET MODE [NO]KEYPAD	Enables or disables keypad mode
DEFINE/KEY	Creates key definitions
DELETE/KEY or UNDEFINE/KEY	Deletes key definitions
{ SET / SHOW } KEY	Establishes the key definition state or displays key definitions
Using Command Procedures and Log Files	
DECLARE	Defines parameters to be passed to command procedures
{ SET / SHOW } LOG	Specifies or identifies the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command
{ SET / SHOW } ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures

Table 4–1 (Cont.): Debugger Command Summary

Command	Description
Using Command Procedures and Log Files	
@file-spec	Executes a command procedure
Using Control Structures	
IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively
WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop
Miscellaneous Commands	
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ LANGUAGE	Establishes or displays the current language
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ EVENT_FACILITY	Establishes or identifies the current run-time facility for language-specific events
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
$\left\{\begin{array}{l}\text{SET}\\\text{SHOW}\end{array}\right\}$ TASK	Modifies the tasking environment or displays task information
$\left\{\begin{array}{l}\text{DISABLE}\\\text{ENABLE}\\\text{SHOW}\end{array}\right\}$ AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled

Using VAX PASCAL Features on the VAX/VMS System

Chapter 5

VAX PASCAL System Environment

This chapter describes the relationship between the VAX/VMS operating system and the VAX PASCAL compiler. It discusses the following topics:

- Program sections
- Storage allocation of variables, constants, and blocks
- Storage allocation sizes of variables of all VAX PASCAL data types
- Alignment boundaries for variables of all VAX PASCAL data types
- Data representation of VARYING OF CHAR and floating-point data

5.1 Program Sections

The VAX PASCAL compiler uses contiguous areas of memory, called program sections, to store information about a program. The VAX/VMS Linker controls memory allocation and sharing according to the properties of each program section. (The *VAX/VMS Linker Reference Manual* refers to the various characteristics of program sections as "attributes." This chapter uses the term "properties" to avoid confusion with the VAX PASCAL attribute classes.) Table 5–1 lists the possible program section properties.

Table 5–1: Program Section Properties

Class	Description
PIC/NOPIC	Position independent or position dependent
CON/OVR	Concatenated or overlaid
REL/ABS	Relocatable or absolute

Table 5–1 (Cont.): Program Section Properties

Class	Description
GBL/LCL	Global or local scope
EXE/NOEXE	Executable or nonexecutable
RD/NORD	Readable or nonreadable
WRT/NOWRT	Writeable or nonwriteable
SHR/NOSHR	Shareable or nonshareable

When constructing an executable image, the linker divides the image into sections. Each image section contains program sections that have the same properties. The linker controls memory allocation by arranging image sections according to program section properties. You can use special linker options to change program section properties 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 options file are described in the *VAX/VMS Linker Reference Manual.*

5.1.1 Using Default Program Sections

The VAX PASCAL compiler can establish up to four program sections, $CODE, $LOCAL, PAS$GLOBAL, and LIB$INITIALIZE. Table 5–2 summarizes the kinds of data contained in these sections by default.

Table 5–2: Program Section Data

Default Program Section	Data
$CODE	Machine instructions; constants needing storage; nonvolatile, read-only, and static variables
$LOCAL	Writeable variables declared with the STATIC attribute; writeable variables that use default static allocation and are declared at program or module level of a nonoverlaid compilation unit
PAS$GLOBAL	Writeable variables that use default allocation and are declared at program or module level of an overlaid compilation unit
LIB$INITIALIZE	Addresses of routines declared with the INITIALIZE attribute

These four default program sections are established only when the program requires them. For example, unless the program contains an overlaid compilation unit, PAS$GLOBAL is not created. If the program does not include any static variables, $LOCAL is not declared. If all necessary program sections are user-defined, the compiler does not establish any of the default program sections.

5.1.2 Using the PSECT Attribute

In a VAX PASCAL program, you can control the allocation of virtual address space by establishing the program section in which storage for a variable, routine, or compilation unit should be allocated. This control is provided by the PSECT attribute. The PSECT attribute has the following form:

```
PSECT(identifier)
```

identifier
The name of the program section. This name can designate either a program section that is established by the compiler or one that is created by the user.

The PSECT attribute is useful for placing static variables and executable blocks in program sections that will be shared among executable images. For example, if several programs access the same error-processing routines, you could group the routines in a separate program section. Because these routines are likely to be called only to handle unusual conditions, the image section containing the executable code for them need not be paged into physical memory until one of the routines is called. Then the image section is paged into main memory until the routine has completed execution.

5.1.3 Using the COMMON Attribute

In a VAX PASCAL program, you can allocate storage for a variable in an overlaid program section called a common block. This capability is provided by the COMMON attribute. The COMMON attribute has the following form:

```
COMMON [[ (identifier) ]]
```

identifier
The name of a common block.

When no identifier is present, the name of the common block is the same as the variable having the COMMON attribute.

Only one variable can be allocated in a particular common block. By storing variables in common blocks, a VAX PASCAL program can share variables with programs written in other VAX languages, such as FORTRAN, PL/I, COBOL, and BASIC.

5.1.4 Establishing Program Section Properties

Whether the compiler establishes a program section, or you create one, the program section is assigned one property from each class listed in Table 5-1. These properties are assigned to satisfy the requirements of the variables and executable blocks that have been allocated in the same program section. Table 5-3 lists the minimal properties required by various objects.

Table 5–3: Required Program Section Properties

Object	Properties			
	EXE	RD	WRT	NOSHR
Read-only Variable		X		
Write-only Variable		X	X	X[1]
Read/write Variable		X	X	X[1]
Executable Block	X	X		

[1]Unless the variable has the COMMON attribute

All program sections except the LIB$INITIALIZE program section are position independent and relocatable; all except those established by the COMMON attribute are concatenated and local. Program sections established by COMMON are overlaid and global. The remaining properties are assigned as follows:

1. The first time the compiler encounters the name of a particular program section (including a common block), it initializes the program section to be readable, nonwriteable, shareable, and nonexecutable. Thus, the program section's initial properties are LCL, NOEXE, NOWRT, CON, PIC, RD, REL, and SHR.
2. If storage for any object not in a common block is allocated in the same program section, the program section instantly becomes writeable and nonshareable. Thus, the program section's write property changes from NOWRT to WRT, and its share property changes from SHR to NOSHR.
3. If storage for a writeable object in a common block is allocated in the same program section, the program section becomes writeable and retains the shareable property. Thus, the program section's write property changes from NOWRT to WRT, and its share property remains SHR.

If you want to guarantee that read-only variables can never be modified, you can allocate storage exclusively for them in a separate program section that will always remain nonwriteable.

5.1.5 Example

The following example uses a common block to pass information between VAX PASCAL and VAX FORTRAN.

VAX PASCAL Program:

```
PROGRAM Common_Example (OUTPUT);
VAR
   Myrec  : [COMMON(Example)] RECORD
      Intfld : INTEGER;
      Strfld : PACKED ARRAY [1..10] OF CHAR;
   END;
[EXTERNAL] PROCEDURE Call_Fort; EXTERNAL;

BEGIN
Myrec := ZERO;
Call_Fort;
WRITELN('Intfld = ',Myrec.Intfld);
WRITELN('Strfld = ',Myrec.Strfld);
END.
```

VAX FORTRAN Subroutine:

```
  SUBROUTINE CALL_FORT
C
  STRUCTURE /TEST/
      INTEGER*4 ITEM
      CHARACTER * 10 ITEM_NAME
  END STRUCTURE
C
  RECORD /TEST/ VAR
  COMMON /EXAMPLE/ VAR
C
  VAR.ITEM = 10
  VAR.ITEM_NAME = '0123456789'
  END
```

The VAX PASCAL program initializes the common record, Myrec, to zero then calls the VAX FORTRAN routine, Call_Fort, to assign the desired values to the elements within the common record. The VAX PASCAL program then writes the assigned values to SYS$OUTPUT.

Recall that only one variable can be allocated in a particular VAX PASCAL common block. To share more than one data item in the same common block, the record variable containing all shareable items is declared and used.

5.2 Storage Allocation

The following sections discuss storage allocation of variables, symbolic constants, and executable blocks. Section 5.2.3 gives examples.

5.2.1 Allocation of Variables

When allocating storage for a variable, the compiler first determines an allocation attribute for the variable. If the allocation attribute is STATIC or COMMON, the compiler then chooses a program section in which to allocate storage. The compiler applies the following rules sequentially to determine the allocation attribute:

1. If the variable is declared with an allocation attribute, the attribute specifies the variable's allocation.
2. If the variable is declared with a visibility attribute other than LOCAL, its allocation is static.
3. If the variable is declared in a routine, its allocation is automatic.
4. If the variable is declared at the outermost level of a module, its allocation is static.
5. If the variable is declared at the outermost level of a program, the compiler must choose between static and automatic allocation. Whenever possible, the compiler uses automatic allocation for variables that are referred to only in the body of the main program because automatic allocation is more efficient. The compiler uses static allocation if the variable is declared with the VOLATILE attribute, initialized at its declaration, or referred to in a nested block, or if the program has an ENVIRONMENT or OVERLAID attribute. Because program-level variables may be statically allocated, VAX PASCAL does not support recursive calls on the main program block.

The compiler applies the following rules sequentially to choose the program section in which to allocate storage for nonexternal common and static variables:

- If the variable has the COMMON attribute, storage is allocated in a common block that has either the same name as the variable, or the name specified by the identifier that accompanies the attribute.
- If the variable has the PSECT attribute, the identifier that accompanies the attribute supplies the name of the program section in which storage is to be allocated.

- If the variable does not have the COMMON, PSECT, or STATIC attribute, but is declared at the outermost level of an overlaid compilation unit, storage is allocated in the program section PAS$GLOBAL.
- If the variable has the READONLY attribute and has neither a PSECT nor a COMMON attribute, its storage is allocated in the program section in which storage for executable code is currently being allocated (see Section 5.2.2).
- All other static variables are allocated in the program section $LOCAL.

5.2.2 Allocation of Symbolic Constants and Executable Blocks

When allocating storage for symbolic constants and executable blocks, the compiler determines the appropriate program section by applying the same rules of scope to program section names that it applies to identifiers. The compiler always allocates storage in the program section whose name appeared in the most recent heading of a routine or compilation unit.

When a program begins, the compiler establishes the $CODE program section. Storage is allocated in $CODE for symbolic constants and executable blocks unless a PSECT attribute appears in the heading of a routine or compilation unit and directs that storage be allocated in a different program section.

If a routine has the INITIALIZE attribute, the address of its entry mask is stored in a program section named LIB$INITIALIZE. The properties of this program section, which include NOPIC, are discussed in the *VAX/VMS Run-Time Library Routines Reference Manual.*

5.2.3 Examples

The following examples illustrate how the compiler establishes program sections to allocate storage for symbolic constants, variables, and executable blocks. The comments in the programs indicate the names of the program sections used.

Example 1

```
PROGRAM Order_Process (INPUT, OUTPUT);

VAR
    { $LOCAL }
    Data_Record : RECORD
        Account_Number : INTEGER;
        Order_Number   : INTEGER;
        Name : VARYING[60] OF CHAR;
        Total_Amount : REAL;
        END;

CONST
     { $CODE }
     Message_String = 'No Invalid Data';

[PSECT(Error_Sect)] PROCEDURE Account_Error;
    VAR
       { $LOCAL }
       Account_Array : [STATIC] ARRAY[1..15] OF INTEGER;

    CONST
        { Error_Sect }
        Error_String = 'Invalid Account Number';

    PROCEDURE Order_Error;
        VAR
            { automatic storage }
            Order_Array : ARRAY[1..10] OF INTEGER

        CONST
            { Error_Sect }
            Error_String = 'Invalid Order Number';

        BEGIN
        { Error_Sect }
        END;

   BEGIN
   { Error_Sect }
   END;

BEGIN
{ $CODE }
END.
```

This example illustrates how the compiler allocates storage in user-created program sections in conjunction with the default VAX PASCAL program sections. Storage for all variables with static allocation is allocated in $LOCAL. Storage for the executable block of the main program and the symbolic constant Message_String is allocated in $CODE. Storage is allocated in the user-created program section, Error_Sect, for the symbolic constant Error_String and the executable block of the procedure Account_Error. Because the procedures Order_Error and Amount_Error fall within the scope of Account_Error, storage is allocated in the same program section, Error_Sect, for their symbolic constants and executable blocks. Storage for Data_Record is in $LOCAL because it was referred to in a nested block.

Example 2

```
[OVERLAID] MODULE Over_Mod;

VAR
   Mod_Var : INTEGER;                    (* PAS$GLOBAL *)
   Real_Var : REAL;                      (* PAS$GLOBAL *)
   Static_Var : [STATIC] INTEGER;        (*   $LOCAL   *)

PROCEDURE Block1
   (VAR Block_Param : REAL);

   VAR
      Block_Var : [STATIC] UNSIGNED;     (* $LOCAL *)

   CONST
      Block_Const = 'Nested Constant';   (* $CODE *)

   BEGIN (* Procedure Block1 *)
   (* $CODE *)

   END;  (* Procedure Block1 *)

END.     (* MODULE Over_Mod *)
```

Because Over_Mod is an overlaid compilation unit, storage is allocated in PAS$GLOBAL for the module-level variables Mod_Var and Real_Var. However, the variables Block_Var and Static_Var are declared with the STATIC attribute; therefore, storage is allocated for them in $LOCAL. Storage is allocated in $CODE for Block_Const and the executable block of the nested procedure Block1.

Example 3

```
MODULE Normal_Mod;

  TYPE
     Read_Int = [READONLY,PSECT(Read_Var)] INTEGER;

  VAR
     Data_File, Answer_File : TEXT;                         (* $LOCAL *)
     Add_Value : Read_Int;                                  (* Read_Var *)
     Mult_Value, Factor : [READONLY,PSECT(Read_Var)] REAL;  (* Read_Var *)
     Div_Value : [READONLY,PSECT(Read_Var)]DOUBLE;          (* Read_Var *)
     In_Value : INTEGER;                                    (* $LOCAL *)
     Result : DOUBLE;                                       (* $LOCAL *)
  END.   (* Module Normal_Mod *)
```

The TYPE section in this example defines the type Read_Int and gives it the READONLY and PSECT attributes. The identifier Read_Var specifies that storage for variables of type Read_Int is to be allocated in the user-created program section Read_Var. The VAR section declares four read-only variables and specifies that storage for them is to be allocated in Read_Var. If no other variables are allocated in Read_Var, the program section will retain the properties NOEXE, NOWRT, RD, and SHR throughout the program. Storage for the other four variables declared in the VAR section is allocated in $LOCAL because they are at the outermost level of a module.

5.3 Allocation Sizes of Variables

The VAX PASCAL compiler allocates storage for variables and components of structured variables when they are declared. For every VAX PASCAL data type, the compiler calculates the allocation size required when a variable of the type occurs in either an unpacked or a packed context. The unpacked size is always represented in bytes, while the packed size is represented in bits.

The packed size of a variable is the minimum number of bits required to represent all values of the variable's type. In general, the compiler uses the "32-bit rule" to determine the allocation size for a component of a structured variable:

- A component whose length is 32 bits or fewer is packed into as few bits as possible and can be unaligned.
- A component whose length is greater than 32 bits is allocated the smallest number of bytes possible and must be aligned on a byte boundary.

If one of the size attributes (BIT, BYTE, WORD, LONG, QUAD, or OCTA) is applied to the variable, the size specified by the attribute represents the variable's packed size. If no size attribute is applied to the variable, the compiler calculates the unpacked size so that it is structurally compatible with the base type of the variable's type.

Table 5-4 illustrates the allocation size for variables of each type when they occur in either a packed or an unpacked context.

Table 5–4: Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
INTEGER	4	32
UNSIGNED	4	32
CHAR	1	8
BOOLEAN	1	1
Enumerated[2]	1, if 256 elements or fewer; 2, if more than 256 elements	log2(number of elements)[1] + 1
Subrange	The size of the base type	The minimum number in which the upper and lower bounds can be expressed[3]
REAL or SINGLE	4	32
DOUBLE	8	64
QUADRUPLE	16	128
Pointer	4	32
Unpacked ARRAY	The sum of the unpacked sizes in bytes of all components, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	Unpacked size in bytes * 8
Unpacked RECORD	The sum of the unpacked sizes in bytes of the fields in the fixed part and the largest variant, plus the sum of the sizes in bytes of any holes created to meet alignment requirements	Unpacked size in bytes * 8

[1]This is known as the ceiling function, where the smallest integer is greater than or equal to X.

[2]The maximum number of elements is 65,535.

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

Table 5–4 (Cont.): Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
PACKED ARRAY	The sum of the packed sizes in bits of all components, plus the sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	The sum of the packed sizes in bits of all components, plus the sum of sizes in bits of any holes created to meet alignment requirements; if the sum is greater than 32, the sum is rounded up to the next multiple of 8
PACKED RECORD	The sum of the packed sizes in bits of all fields in the fixed part and the largest variant, plus the sum of sizes in bits of any holes created to meet alignment requirements; this sum is rounded up to a multiple of 8 and then divided by 8	The sum of the packed sizes in bits of all fields in the fixed part and the largest variant, plus the sum of sizes in bits of any holes created to meet alignment requirements; if the sum is greater than 32, the sum is rounded up to the next multiple of 8
VARYING OF CHAR	Maximum length + 2	(Maximum length + 2) * 8
Unpacked SET[3]	32, if the set base type is subrange of INTEGER or UNSIGNED; else, compute (ORD(upper-bound of ordinal type that is base type of set's base type) + 8) DIV 8, and if this result is less than or equal to 8, the result is rounded up to 1, 2, 4, or 8	Compute (ORD (upper-bound) + 1); if the result is less than or equal to 64, the result is rounded up to 8, 16, 32, or 64, and if the result is greater than 64, the result is rounded to next higher multiple of 8

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

Table 5–4 (Cont.): Storage of Types

Data Type	Unpacked Size in Bytes	Packed Size in Bits
PACKED SET[3]	The result of (ORD(upper-bound) + 8) DIV 8	Compute (ORD (upper-bound) + 1); if the result is greater than 32, the result is rounded to next higher multiple of 8
FILE	Not specified	Not specified

[3]Sets of type INTEGER and UNSIGNED are limited to 256 bits.

NOTE

The formula to compute the size of a packed subrange is MAX (X,Y) + Z where X, Y, and Z are computed as follows (LOW represents the low bound of the subrange; HIGH represents the upper bound):

```
IF LOW < -1                          IF HIGH > 0
THEN                                 THEN
   X := log2(-LOW - 1) + 1              Y := log2(HIGH) + 1
ELSE                                 ELSE
   X := 0;                              Y := 0;

IF LOW >= 0
THEN
   Z := 0
ELSE
   Z := 1;
```

You can discover both the unpacked and packed sizes for variables of any type by using the predeclared functions BITNEXT, BITSIZE, NEXT, and SIZE. These functions, which are fully described in the *VAX PASCAL Reference Manual,* require a parameter that is the name of either a type or a variable:

- BITNEXT returns an integer value that indicates what the packed size would be for an array component of the type.
- BITSIZE returns an integer value that indicates what the packed size would be for a record field of the type.
- NEXT returns an integer value that indicates what the unpacked size would be for an array component of the type.

- SIZE returns an integer value that indicates what the unpacked size would be for a variable or record field of the type.

5.3.1 Examples

The following examples illustrate the effects of packing records and multidimensional arrays at various levels.

Example 1

```
TYPE
   Internal_Arr = ARRAY[1..5] of 0..6;

VAR
   Samp1_Arr : PACKED ARRAY[1..5] OF Internal_Arr;
```

Each component of an array of type Internal_Arr is stored in a longword. Each component of Samp1_Arr, in turn, requires five longwords—enough storage space for five components of type Internal_Arr. The entire array Samp1_Arr therefore occupies 25 longwords (800 bits).

Example 2

```
VAR
   Samp1_Arr : ARRAY[1..5] OF PACKED ARRAY[1..5] OF 0..6;
```

Each PACKED ARRAY[1..5] of 0..6 requires 15 bits. Because the packed arrays are components of an unpacked array, their size is rounded up to an even 16 bits. The total size of Samp1_Arr is therefore 80 bits.

Example 3

```
TYPE
   Internal_Arr = PACKED ARRAY[1..5] OF 0..6;

VAR
   Samp2_Arr : PACKED ARRAY[1..5] OF Internal_Arr;
   Samp3_Arr : PACKED ARRAY[1..5,1..5] OF 0..6;
```

In this example, every component of Internal_Arr requires only three bits because the array is packed. Each component of Samp2_Arr and Samp3_Arr can be stored in 15 bits, and each array occupies 75 bits. Except when the program is compiled with the /OLD_VERSION qualifier (see the *VAX PASCAL Reference Manual*), the specification of PACKED for an array with multiple indexes results in packing at every level. Therefore, the two arrays in this example are equivalent.

Example 4

```
VAR
   Sample : PACKED ARRAY[1..5,1..5,1..5] OF 0..6;
```

This example shows space savings for arrays of more than two dimensions when PACKED is specified at every level. The subrange 0..6 requires 3 bits; five 3-bit components require 15 bits. This size describes the innermost dimension of Sample. Next, five 15-bit components require 75 bits. Because of the 32-bit rule, each 75-bit component is rounded up to 80 bits. This size describes the middle and inner dimensions of Sample. Finally, five 80-bit components require 400 bits (50 bytes). The entire array Sample, then, requires 400 bits.

Example 5

```
VAR
   Sample_Rec : PACKED RECORD
                   Field_1 : BOOLEAN;
                   Field_2 : INTEGER;
                   Field_3 : DOUBLE;
                END;
```

In this example, Field_1 requires only 1 bit of storage. Field_2 is 32 bits in size, and starts immediately following Field_1. Because Field_3 is larger than 32 bits, it will start on the next byte boundary. The entire record Sample_Rec, therefore, requires 104 bits.

5.4 Alignment Boundaries

The memory-addressing boundary on which a variable or a component is aligned depends on the variable's or the component's allocation size. You can change the alignment by using the ALIGNED and UNALIGNED attributes, as explained in the *VAX PASCAL Reference Manual.* The following conditions determine boundary alignment:

- If the variable or component has an alignment attribute, the compiler uses the specified alignment. You cannot apply the UNALIGNED attribute to a variable or component whose allocation size is greater than 32 bits. A variable or component of this size must be aligned on a byte boundary.
- Variables declared without an alignment attribute are aligned by default on a byte boundary. The compiler can align such variables on a larger storage boundary if it can access them more efficiently by doing so.

- All components of an unpacked array or record variable are byte aligned within the array or record. The array or record itself, however, may be unaligned.
- The alignment of components of a packed array, packed record, or VARYING OF CHAR variable always follows the 32-bit rule (see Section 5.3).
- Dynamic variables allocated by the NEW procedure are always aligned on a quadword boundary.

Although you can save storage space by packing variables of structured types, you must be careful to pack at the proper level. Except for its alignment, a record field whose type is an unpacked array, set, or record occupies the same amount of space in a packed or an unpacked record variable. To pack such a field, you must explicitly declare its type to be packed.

When packing multidimensional arrays, you must specify packing at the innermost level to gain any significant space advantage. For example, there is no advantage in packing an array of an unpacked structured type—the unpacked components will still be aligned on byte boundaries, thereby leaving holes in the storage space. To gain storage space, you must specify a packed array of a packed structured type.

5.4.1 Examples

The following examples show the use of alignment boundaries.

Example 1

```
VAR
   X : [STATIC, ALIGNED (3)] INTEGER;  (* $LOCAL, QUADWORD ALIGNED *)
```

In this example, X is declared a static variable that is aligned on a QUADWORD boundary.

Example 2

```
VAR
   Page_Name : [PSECT(New_Section), ALIGNED(9)] PACKED ARRAY [1..512] OF
      [BYTE] 0..255;             (* New_Section, PAGE ALIGNED *)
```

This example declares the page aligned variable Page_Name, which is to be allocated in the program section New_Section.

Example 3

```
VAR
   X : PACKED RECORD                        (* AUTOMATIC *)
             Field1 : BOOLEAN;
             Field2 : REAL;
             Field3 : BOOLEAN;
             Field4 : DOUBLE;
             END;
```

In this example, Field1 begins at bit position 0 and is 1 bit long. Field 2 begins at bit position 1 and is 32 bits long. Field3 begins at bit position 33 and is 1 bit long. However, because Field4 is greater than 32 bits long (DOUBLE requires 64 bits) Field4 must be byte aligned. For this reason, Field4 begins on bit position 40 and is 64 bits long. See Section 5.3 for the 32-bit rule.

Example 4

```
VAR
   X : RECORD                            (* AUTOMATIC *)
       Field1 : [BIT(3)]  0..7;
       Field2 : [UNALIGNED] INTEGER;
       END;
```

In this example, Field1 of record X is declared to begin on bit position 0 and is 3 bits long. Due to the use of the UNALIGNED attribute on Field2, Field2 begins on bit position 3 and is 32 bits long. Note that you can obtain the same behavior by packing record X.

Without using the UNALIGNED attribute, or without X being a PACKED record, Field2 would have been byte aligned; that is, it would have started on bit position 8.

5.5 Representation of Varying Data

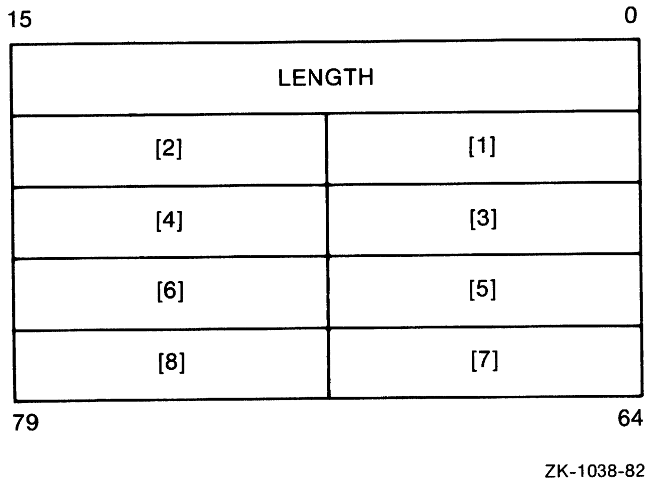
A variable of type VARYING OF CHAR is stored as though it were a VAX PASCAL record type of the following form:

```
RECORD
Length : [WORD] 0..Maxlength;
Body : PACKED ARRAY[1..Maxlength] OF CHAR;
END;
```

Storage is allocated as one byte per character, with an initial field of two bytes to indicate the total length. A variable of type VARYING OF CHAR whose length is greater than or equal to three characters is allocated an exact number of bytes. Storage allocation for a variable of type VARYING OF CHAR whose maximum length is zero, one, or two characters follows the 32-bit rule in that the variable can have the UNALIGNED attribute (see Section 5.3).

Figure 5–1 illustrates the storage allocated for a variable of type VARYING[8] OF CHAR.

Figure 5–1: Storage of Varying Data



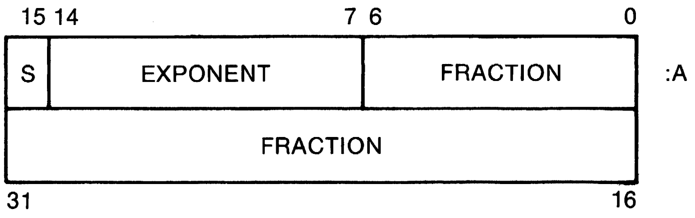
5.6 Representation of Floating-Point Data

The following sections summarize the internal representation of single-precision (REAL and SINGLE types), double-precision (DOUBLE type), and quadruple-precision (QUADRUPLE type) floating-point numbers. For more detailed information, see the *Introduction to VAX/VMS System Routines.*

5.6.1 Single-Precision (SINGLE and REAL Types)

A single-precision floating-point number is represented by four contiguous bytes. The bits are numbered from the right, 0 through 31, as shown in Figure 5–2.

Figure 5–2: Single-Precision Floating-Point Data Representation



ZK-1039-82

A single-precision floating-point value is specified by its address A, the address of the byte containing bit 0. The form of an F_floating-point value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 7 are an excess 128 binary exponent.
- Bits 6 through 0 and 31 through 16 are 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 from 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 number 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 considered a reserved operand. Floating-point instructions that process a reserved operand cause a reserved operand fault.

On the VAX, 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.

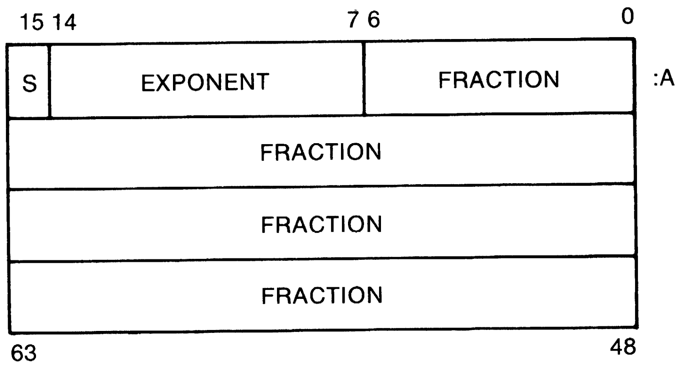
5.6.2 Double-Precision (DOUBLE Type)

A double-precision floating-point value is represented by eight contiguous bytes. Variables of type DOUBLE take one of two formats: D_floating or G_floating. D_floating format allows values to be expressed with greater precision than that allowed by the G_floating format; G_floating format allows a wider range of values to be expressed than that allowed by the D_floating format.

5.6.2.1 D_Floating-Point

D_floating-point data is stored in the format shown in Figure 5-3. The bits are numbered from the right, 0 through 63.

Figure 5-3: D_Floating-Point Double-Precision Representation



ZK-1040-82

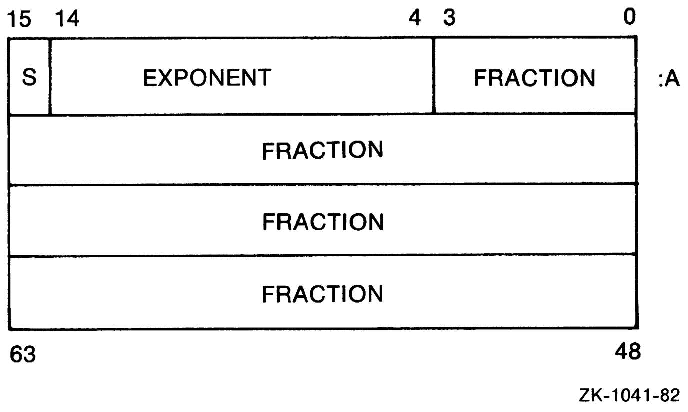
A D_floating-point double-precision value is specified by its address A, the address of the byte containing bit 0. The form of a D_floating-point value is identical to that of a single-precision floating-point value except for an additional 32 low-significance fraction bits. Within the fraction, bits 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 D_floating-point values as for single-precision floating-point values. The precision of a D_floating-point value is approximately one part in 2^{55}, or 16 decimal digits.

5.6.2.2 G_Floating-Point

G_floating-point data is stored in the format shown in Figure 5–4. The bits are numbered from the right, 0 through 63.

Figure 5–4: G_Floating-Point Double-Precision Representation



A G_floating-point double-precision value is specified by its address A, the address of the byte containing bit 0. The form of a G_floating-point value is sign magnitude as follows:

- Bit 15 is the sign bit.
- Bits 14 through 4 are an excess 1024 binary exponent.
- Bits 3 through 0 and 63 through 16 represent a normalized 53-bit fraction without the redundant most significant fraction bit. Within the fraction, bits of increasing significance go from 48 through 63, 32 through 47, 16 through 31, and 0 through 3.

The 11-bit exponent field encodes the values 0 through 2047:

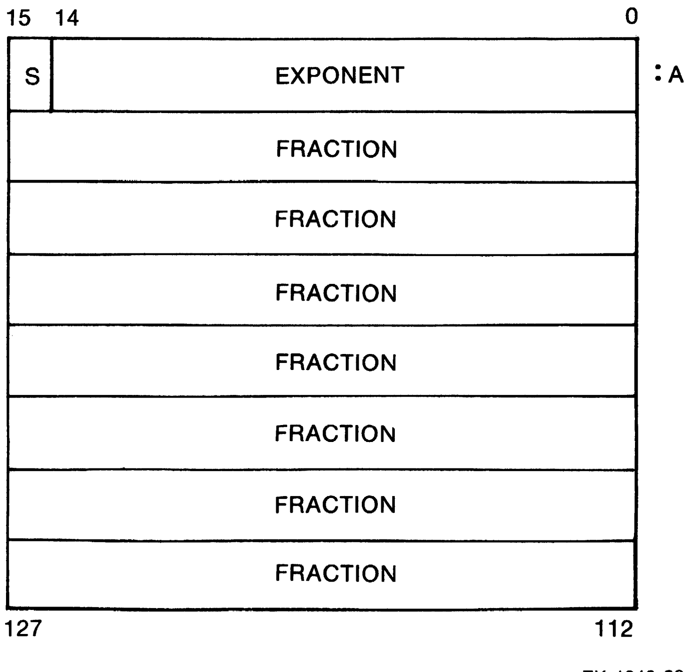
- An exponent value of 0, with a sign bit of 0, indicates that the G_floating-point number has a value of 0.
- Exponent values of 1 through 2047 indicate binary exponents of −1023 through +1023.
- An exponent value of 0, together with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The value of a G_floating-point number is in the approximate range of .56 * (10^{-308}) through .90 * (10^{308}). The precision of a G_floating-point value is approximately one part in 2^{52}, or 15 decimal digits.

5.6.3 Quadruple-Precision (QUADRUPLE Type)

A quadruple-precision floating-point value is represented by 16 contiguous bytes. The bits are numbered from the right 0 through 127, as shown in Figure 5–5.

Figure 5–5: Quadruple-Precision Floating-Point Representation



A quadruple-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 as follows:

- Bit 15 is the sign bit.
- Bits 14 through 0 are an excess 16,384 binary exponent.
- Bits 127 through 16 are a normalized 113-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of increasing significance go from 112 through 127, 96 through 111, 80 through 95, 64 through 79, 48 through 63, 32 through 47, and 16 through 31.

The 15-bit exponent field encodes the values 0 through 32,767:

- An exponent value of 0, with a sign bit of 0, indicates that the quadruple-precision number has a value of 0.
- Exponent values of 1 through 32,767 indicate true binary exponents of –16,383 through +16,383.
- An exponent value of 0, with a sign bit of 1, is considered a reserved operand. Floating-point instructions processing a reserved operand cause a reserved operand fault.

The value of a quadruple-precision floating-point number is in the approximate range of $0.84 * (10^{-4932})$ through $0.59 * (10^{4932})$. The precision of a quadruple-precision value is approximately one part in 2^{112}, or 33 decimal digits.

Chapter 6

Input and Output with RMS

This chapter provides information about VAX PASCAL I/O in relation to VAX Record Management Services (RMS). RMS is the system VAX PASCAL uses to transfer data between an input or output device and a VAX PASCAL program. Note that RMS uses the term "record" to denote a file component; this chapter uses the term "component" whenever possible to avoid confusion with the VAX PASCAL record data structure. The following topics are discussed in this chapter:

- RMS file characteristics
- I/O error recovery
- Syntax for the OPEN and CLOSE procedures
- File sharing
- RMS record locking
- Use of indexed files
- Programming considerations involving input and output
- Interprocess communication by means of mailboxes
- Remote communication by means of DECnet-VAX

6.1 RMS File Characteristics

The following sections describe the elements of VAX PASCAL I/O processing: RMS records, files, and access methods.

6.1.1 RMS Record Format

Record format refers to the way the RMS record physically appears on a storage medium. RMS records are stored in one of the following formats:

- Fixed-length format
- Variable-length format
- Stream format

You specify the format of a file's components with the record type parameter of the OPEN procedure, as described in Section 6.3. You can use either fixed or variable formats with any file organization; however, stream record format is supported only for sequential files.

6.1.1.1 Fixed-Length Record Format

In a file that has fixed-length RMS records, all file components contain the same number of bytes. When you create a file that is to contain fixed-length records, the maximum size of the file components is used as the record length. This maximum size can be the default size of the file components, or it can be specified with the record length parameter of the OPEN procedure (see Section 6.3). A file with sequential organization opened for direct access must contain fixed-length records, to allow the component number to be computed correctly.

6.1.1.2 Variable-Length Record Format

In a file that has variable-length RMS records, the file components can contain any number of bytes, up to a specified maximum. Each component is prefixed by a count field whose value indicates the number of data bytes in that component. The count field itself requires two bytes on a disk device and four bytes on magnetic tape.

Variable-length records in relative files are actually stored in fixed-length cells. The maximum size of the file components is used as the length of the cells. This maximum size can be the default size of the components, or it can be specified with the record length parameter of the OPEN procedure (see Section 6.3). The value of this parameter indicates the size of the largest component that can be stored in the file.

6.1.1.3 Stream Record Format

Stream format is an RMS record format in which RMS records in a file are delimited by special characters or character sequences called terminators. Terminators are part of the RMS record they delimit. The data in a stream format file is interpreted as a continuous sequence of bytes, without control information such as record counts, segment flags, or other system-supplied boundaries. Stream format is supported for sequential files only. STREAM_CR and STREAM_LF indicate carriage return and line feed, respectively. For more information on stream format, see the *VAX Record Management Services Reference Manual.*

6.1.2 RMS File Organization

File organization refers to the way RMS records are physically arranged on a storage device. File organization is specified when the file is created, and cannot be changed. VAX PASCAL supports three RMS file organizations:

- Sequential
- Relative
- Indexed

You specify the organization of a file with the organization parameter of the OPEN procedure, as described in Section 6.3.

6.1.2.1 Sequential Organization

A sequential file consists of components arranged in the sequence in which they are written to the file: the first component written is the first component in the file, the second component written is the second component in the file, and so on. As a result, components can be added only at the end of the file.

Sequential file organization is permitted on all devices supported by RMS.

6.1.2.2 Relative Organization

A relative file consists of numbered positions, called cells. These cells are of fixed equal length and are numbered consecutively from 1 to n, where 1 is the first cell available in the file and n is the last.

With this organization, you place a component in a file according to its cell number. The cell number is also the component's relative component number, that is, the location of the component relative to the beginning of the file. As a result, you can retrieve a component directly by specifying its relative component number.

You can add a component to, or delete one from, a file regardless of its location as long as you keep track of its relative component number.

Relative files are supported only on disk devices.

6.1.2.3 Indexed Organization

In an indexed file, each component includes one or more key fields. Each key field establishes an ordering of the file components. Each component in an indexed file must contain at least one key. This mandatory key, called the primary key, determines the location of a component within the body of the file.

The keys of all components are collected to form one or more indexes, through which components are always accessed. The structure of the indexes allows a program to access components in an indexed file either randomly, by specifying particular key values, or sequentially, by retrieving components with increasing or decreasing key values. In addition, keyed access and sequential access can be mixed. The term Indexed Sequential Access Method (ISAM) refers to this dynamic access feature.

Indexed files are supported only on disk devices. See Section 6.8 for more information on indexed files.

6.1.3 RMS Record Access

Record access refers to the method used to read components from or write components to a file, regardless of the file's organization. Record access is specified each time the file is opened and can vary. VAX PASCAL supports three RMS access methods:

- Sequential
- Direct
- Keyed

You specify the access method of a file with the access-method parameter of the OPEN procedure, as described in Section 6.3.

Your choice of access method is affected by the organization of the file to be accessed. For example, sequential access can be used with sequential, relative, and indexed files; but keyed access can be used only with indexed files.

Table 6–1 shows the valid combinations of access method and file organization.

Table 6–1: Valid Combinations of Record Access Method and File Organization

File Organization	Access Method		
	Sequential	**Direct**	**Keyed**
Sequential	Yes	Yes[1]	No
Relative	Yes	Yes	No
Indexed	Yes	No	Yes

[1]Fixed-length records only.

6.1.3.1 Sequential Access

If you select sequential access for a file with sequential or relative organization, a particular component can be read only after all the components that precede it have been read; a new component can be written only at the end of the file.

With an indexed file opened for sequential access, components are read and written according to the calling sequence of the primary key values. Depending on the collating sequence, reading from an indexed file retrieves the next component with the same, next higher, or next lower specified key value.

6.1.3.2 Direct Access

If you select direct access, you can determine the order in which components are read or written. Each series of calls to the READ procedure must be preceded by a FIND or RESET procedure, which positions the file at the specified component. Each series of calls to the WRITE procedure must be preceded by a LOCATE or REWRITE procedure.

You can use direct access on relative files and on sequential disk files that contain fixed-length RMS records. Because the direct access method uses component numbers to find file components, you can execute successive FIND or LOCATE procedures that request components in any order you choose. For example:

```
FIND (Order_File, 24);
READ (Order_File, Order_Amt);
FIND (Order_File, 10);
READ (Order_File, Order_Amt);
```

These successive statements read component 24 and then component 10.

A file for which you select direct access can be accessed sequentially as well, using GET and PUT procedures (see the *VAX PASCAL Reference Manual* for more information).

6.1.3.3 Keyed Access

If you select keyed access, you can determine the order in which components are read or updated by manipulating the keys of the components using the FINDK and RESETK procedures. The key value parameter you specify with each FINDK or RESETK procedure call is compared with index entries until the desired component is located.

When you insert a new component, the value contained in the key field of that component determines its placement in the file.

You can use keyed access only for indexed files. A file for which you select keyed access can be accessed sequentially as well, using GET and PUT procedures. (see the *VAX PASCAL Reference Manual* for more information).

6.2 Input/Output Error Detection

VAX PASCAL's I/O procedures accept an optional parameter called ERROR. The error recovery parameter specifies the action the program should take if the procedure fails to execute successfully. The ERROR parameter can have one of two values, CONTINUE or MESSAGE.

When you specify ERROR := CONTINUE, the program continues to execute regardless of most error conditions encountered during execution of the procedure. If you specify ERROR := MESSAGE, an appropriate error message will be generated and execution will cease if the procedure results in an error. By default, VAX PASCAL generates an error message and ceases execution after it encounters the first error.

When you use the ERROR := CONTINUE parameter with a procedure that operates on a file, you should then use the STATUS function to determine whether execution of the procedure resulted in an error. STATUS returns an integer value to indicate the effect of the last operation on a file. A value of 0 indicates a successful operation; a value of –1 indicates that the previous operation encountered an end-of-file; a positive integer value indicates the specific error that resulted from the previous operation. (See Appendix B for a list of the specific error condition codes returned.)

The ERROR parameter cannot be used with the functions EOF, EOLN, or UFB, nor can it be used with references to the file buffer (f^). Therefore, you should call STATUS to check the status of the file before calling one of these functions or making a reference to the file buffer. If STATUS returns the value 0, the operation can be executed safely.

Note that the STATUS function, when used on text files, causes delayed device access to occur, resulting in the filling of the file buffer (see Section 6.9.4). Because of delayed device access, unexpected results can occur if you use the STATUS function following a READLN procedure.

Remember that a READLN procedure call performs a READ procedure on each variable listed as a parameter, then performs a READLN procedure to position the file at the beginning of the next line. Therefore, a call to STATUS after a READLN procedure tests whether the file was successfully positioned. To test the status of the file, STATUS fills the file buffer with the next component by performing delayed device access. If you want to test the successful reading of data from the input file, you should read the data with the READ procedure, call the STATUS function, and then perform a READLN procedure to advance the file to the beginning of the next line.

6.3 OPEN Procedure Syntax

The OPEN procedure opens a file, defines the file access method, and allows you to specify file parameters. The term "record" in the parameter names of the OPEN procedure indicates an RMS record. The OPEN procedure can have one of the following two formats:

```
1.   OPEN (file-variable
          ,⟦file-name⟧
          ,⟦history⟧
          ,⟦record-length⟧
          ,⟦access-method⟧
          ,⟦record-type⟧
          ,⟦carriage-control⟧
          ,⟦organization⟧
          ,⟦disposition⟧
          ,⟦file-sharing⟧
          ,⟦user-action⟧
          ,⟦default-file-name⟧
          ⟦,ERROR := error-recovery⟧)

2.             FILE_VARIABLE := file-variable
             / ⟦,FILE_NAME := file-name⟧                    \
             | ⟦,HISTORY := history⟧                        |
             | ⟦,RECORD_LENGTH := record-length⟧            |
             | ⟦,ACCESS_METHOD := access-method⟧            |
             | ⟦,RECORD_TYPE := record-type⟧                |
             | ⟦,CARRIAGE_CONTROL := carriage-control⟧      |
     OPEN (  < ⟦,ORGANIZATION := organization⟧              >...)
             | ⟦,DISPOSITION := disposition⟧                |
             | ⟦,SHARING := file-sharing⟧                   |
             | ⟦,USER_ACTION := user-action⟧                |
             | ⟦,DEFAULT := default-file-name⟧              |
             \ ⟦,ERROR := error-recovery⟧                   /
```

Except for the file variable, all parameters are optional and are RMS-dependent properties. Table 6–2 summarizes the parameters and their defaults.

If the parameter names are not specified, as in format 1, the parameters must be listed in the specified order. If parameter names are specified, as in format 2, the parameters can be specified in any order. You can mix the use of positional and nonpositional parameters, but once a nonpositional parameter name has been used, all the following parameter values must be nonpositional.

Table 6–2: Summary of OPEN Procedure Parameters

Parameter	Parameter Values	Default
File variable	Any file type	None; the file variable is a required parameter.
File name	Any character string	File-variable name expression
History	OLD, NEW, READONLY, UNKNOWN	NEW (OLD, if an external file is opened using RESET.)
Record length	Any positive integer value	For VAX/VMS Version 4.6 or higher, the default is 255 bytes; otherwise, the default is 133 bytes for text files; for other files, this parameter is ignored.
Access method	DIRECT, KEYED, or SEQUENTIAL	SEQUENTIAL.
Record type	FIXED, VARIABLE, STREAM, STREAM_CR, STREAM_LF	VARIABLE for new text files and FILE OF VARYING; FIXED for other new files; for old files, the record type is established at file creation.
Carriage control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, NONE	LIST for text files and FILE OF VARYING; NOCARRIAGE for all other files. Old files use their existing carriage-control parameters.
Organization	SEQUENTIAL, RELATIVE, INDEXED	SEQUENTIAL for new files; the previous organization for existing files.
Disposition	SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, SUBMIT_DELETE	SAVE for named files; DELETE for files without a file-name parameter.

Table 6–2 (Cont.): Summary of OPEN Procedure Parameters

Parameter	Parameter Values	Default
Sharing	READONLY, READWRITE, NONE	READONLY if the file history is READONLY; NONE for all other files.
User action	Function-identifier	None.
Default file name	Any character string expression	None.
Error recovery	CONTINUE, MESSAGE	MESSAGE.

For more information on the OPEN procedure see the *VAX PASCAL Reference Manual.*

6.4 CLOSE Procedure Syntax

The CLOSE procedure closes an open file and can have one of the following forms:

```
1.   CLOSE (file-variable
            ,[[disposition]]
            ,[[user-action]]
            [[,ERROR := error-recovery]])

2.          FILE_VARIABLE := file-variable
           / [[,DISPOSITION := disposition]] \
   CLOSE( <  [[,USER_ACTION := user-action]]  >    ...)
           \ [[,ERROR := error-recovery]]    /
```

Table 6–3: Summary of CLOSE Procedure Parameters

Parameter	Parameter Values	Default
File variable	Any file type	None; the file variable is a required parameter.
Disposition	SAVE, DELETE, PRINT, PRINT_DELETE, SUBMIT, SUBMIT_DELETE	SAVE for named files; DELETE for files without a file-name parameter on the corresponding OPEN.
User action	Function-identifier	None.
Default file name	Any character string expression	None.
Error recovery	CONTINUE, MESSAGE	MESSAGE.

Except for the file variable, all parameters are optional and are RMS-dependent properties. For more information about the CLOSE procedure, see the *VAX PASCAL Reference Manual.*

6.5 User Action Functions

The user action parameter of the OPEN procedure allows you to access RMS facilities not explicitly available in the VAX PASCAL language by writing a function that controls the opening of the file. Inclusion of the user action parameter causes the Run-Time Library to call your function to open the file instead of calling RMS to open it according to its normal defaults.

The user action parameter of the CLOSE procedure is similar to that of the OPEN procedure. It allows you to access RMS facilities not directly available in VAX PASCAL by writing a function that controls the closing of the file. Including the user action parameter causes the Run-Time Library to call your function to close the file instead of calling RMS to close it according to its normal defaults.

When an OPEN or CLOSE procedure is executed, the Run-Time Library uses the procedure's parameters to establish the RMS file access block (FAB) and the record access block (RAB), as well as to establish its own internal data structures. These blocks are used to transmit requests for file and record operations to RMS; they are also used to return the data contents of files, information about file characteristics, and status codes.

In order, the three parameters passed to a user action function by the Run-Time Library are as follows:

1. FAB address
2. RAB address
3. File variable

A user action function is usually written in VAX PASCAL and includes the following:

- Modifications to the FAB or RAB, or both (optional)
- $OPEN and $CONNECT for existing files or $CREATE and $CONNECT for new files (required)
- Status check of the values returned by $OPEN or $CREATE and $CONNECT (required)
- Storage of FAB and RAB values in program variables (optional)
- Return of success or failure status value for the user action function (required)

NOTE

Modification of any of the RMS file access blocks provided by the RTL may interfere with the normal operation of the VAX PASCAL Run-time Library.

Example

The following example shows a VAX PASCAL program that copies one file into another. The program features two user action functions, which allow the output file to be created with the same size as the input file and to be given contiguous allocation on the storage media.

```
[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM Contiguous_Copy (F_In, F_Out);

(* The input file F_In is copied to the output file F_Out.
   F_Out has the same size as F_In and has contiguous
   allocation. *)


TYPE
   FType = FILE OF VARYING[133] OF CHAR;

VAR
   F_In, F_Out : FType;
   Alloc_Quantity : UNSIGNED;
```

```
FUNCTION User_Open
   (VAR FAB : FAB$TYPE;
    VAR RAB : RAB$TYPE;
    VAR F   : FType)    : INTEGER;

   VAR
     Status : INTEGER;

   BEGIN          (* Function User_Open *)
   (* Open file and remember allocation quantity *)
   Status := $OPEN (FAB);
   IF ODD (Status)
   THEN
      Status := $CONNECT (RAB);
   Alloc_Quantity := FAB.FAB$L_ALQ;
   User_Open := Status;
   END;           (* Function User_Open *)


FUNCTION User_Create
   (VAR FAB : FAB$TYPE;
    VAR RAB : RAB$TYPE;
    VAR F   : FType)    : INTEGER;

   VAR
     Status : INTEGER;

   BEGIN          (* Function User_Create *)
   (* Set up contiguous allocation *)
   FAB.FAB$L_ALQ := Alloc_Quantity;
   FAB.FAB$V_CBT := FALSE;
   FAB.FAB$V_CTG := TRUE;
   Status := $CREATE (FAB);
   IF ODD (Status)
   THEN
        Status := $CONNECT (RAB);
   User_Create := Status;
   END;           (* Function User_Create *)


BEGIN             (* main program *)
(* Open files *)
OPEN (F_In,
      HISTORY := READONLY,
      USER_ACTION := User_Open);
RESET (F_In);
OPEN (F_Out,
      HISTORY := NEW,
      USER_ACTION := User_Create);
REWRITE (F_Out);


(*Copy F_In to F_Out*)
WHILE NOT EOF (F_In) DO
   BEGIN
   WRITE (F_Out, F_In_^);
   GET (F_In);
   END;
```

```
(* Close files *)
CLOSE (F_In);
CLOSE (F_Out);
END.              (* main program *)
```

In this example, the record types FAB$TYPE and RAB$TYPE are defined in SYS$LIBRARY:STARLET, which the program inherits. The function User_Open is called as a result of the OPEN procedure for the input file F_In. The function begins by opening the file with the RMS service $OPEN. If $OPEN succeeds, the value of Status is odd; in that case, $CONNECT is performed. The allocation quantity contained in the FAB.FAB$L_ALQ field of the FAB is assigned to a variable so that this value can be used in the second user action function. User_Open is then assigned the value of Status (in this case, TRUE), which is returned to the main program.

The function User_Create is called as a result of the OPEN procedure for the output file F_Out. The function assigns the allocation quantity of the input file to the FAB.FAB$L_ALQ field of the FAB, which contains the allocation size for the output file. The FAB field FAB.FAB$V_CBT is set to FALSE to disable the request that file storage be allocated contiguously on a best try basis. Then, the FAB field FAB.FAB$V_CTG is set to TRUE so that contiguous storage allocation is mandatory. Finally, the RMS service $CREATE is performed. If $CREATE is successful, $CONNECT will be done and the function return value will be that of $CREATE.

Once the OPEN procedures have been performed successfully, the program can then accomplish its main task, copying the input file F_In to the output file F_Out, which is the same size as F_In and has contiguous allocation. The last step in the program is to close the files.

6.6 File Sharing

Through the RMS file sharing capability, a file can be accessed by more than one open program at a time or by the same program through more than one file variable. There are two kinds of file sharing—read sharing and write sharing. Read sharing occurs when several programs are reading a file at the same time. Write sharing takes place when at least one program is writing a file and at least one other program is either reading or writing the same file.

The extent to which file sharing can take place is determined by three factors: the type of device on which the file resides, the organization of the file, and the explicit information supplied by the user. These elements affect file sharing in the following ways:

- Device type

  Sharing is possible only on disk files, since other files must be accessed sequentially.

- File organization

  All three file organizations permit read and write sharing on disk files.

- Explicit user-supplied information

  Whether or not file sharing actually takes place depends on two items of information that you provide for each program accessing the file. In VAX PASCAL programs, this information is supplied by the values of the SHARING and HISTORY parameters in the OPEN procedure.

The HISTORY parameter determines how the program will access the file. HISTORY := NEW, HISTORY := OLD, and HISTORY := UNKNOWN determine that the program will read from and write to the file. HISTORY := READONLY determines that the program will only read from the file. If you try to open an existing file with HISTORY := OLD or HISTORY := UNKNOWN, the Run-time Library retries the OPEN procedure with HISTORY := READONLY if the initial OPEN fails with a privilege violation.

The SHARING parameter determines what other programs are allowed to do with the file. Read sharing can occur when SHARING := READONLY is specified by all programs that access the file. Write sharing is accomplished when all programs specify SHARING := READWRITE. To prevent sharing, specify SHARING:= NONE with the first program to access the file.

Programs that specify SHARING := READONLY or SHARING := READWRITE can access a file simultaneously; however, file sharing can fail under certain circumstances. For example, a program without either of these parameters will fail when it attempts to open a file currently being accessed by some other program. Or, a program that specifies SHARING := READONLY or SHARING := READWRITE can fail to open a file because a second program with a different specification is currently accessing that file.

When two or more programs are write sharing a file, each program should use one of the error-processing mechanisms described in Chapter 7. Use of such controls prevents program failure due to a record-locking error. Section 6.7 discusses error detection and the handling of locked records.

6.7 Record Locking

The RMS record locking facility, along with the logic of the program, prevents two processes from accessing the same component simultaneously. It ensures that a program can add, delete, or update a component without having to do a synchronization check to determine whether that component is currently being accessed by another process.

When a program opens a relative or indexed file and specifies SHARING := READWRITE, RMS locks each component as it is accessed. When a component is locked, any program that attempts to access it fails and a record-locked error results. A subsequent I/O operation on the file variable unlocks the previously accessed component. Thus, at most one component is locked for each file variable.

A VAX PASCAL program can explicitly unlock a component by executing the UNLOCK procedure. To minimize the time during which a component is locked against access by other programs, the UNLOCK procedure should be used in programs that retrieve components from a shared file but that do not attempt to update them. VAX PASCAL requires that a component be locked before a DELETE or an UPDATE procedure can be executed.

6.8 Indexed Files

You can access indexed files with both sequential and keyed access methods. Sequential reading retrieves consecutive components, which are sorted according to the specified key field and the collating sequence. Keyed access permits random component selection according to the value of a particular key field. Once you select a component by key, a sequential read retrieves components with ascending or descending key values depending on the key's collating sequence, beginning with the key field value of the initial FINDK or RESETK procedure. When you specify the KEYED access method in the OPEN procedure, you enable both sequential and keyed access to the indexed file.

Indexed organization is especially suitable for manipulating complex files in which you want to select components based on one of several criteria. For example, a mail-order firm could use an indexed file to store its customer list. Key fields might designate a unique customer order number, the customer's zip code, and the item ordered. Reading sequentially based on the zip code key would produce a mailing list already sorted by zip code, while reading sequentially based on the item-ordered key would give a list of customers sorted according to the product ordered.

6.8.1 Creating an Indexed File

Within a VAX PASCAL program, you can create an indexed file in two ways:

- With the VAX PASCAL OPEN procedure
- With the RMS file definition utility ($CREATE/FDL)

You can use the OPEN procedure to specify common file characteristics; you must use the $CREATE/FDL utility to select features not directly supported by VAX PASCAL. However, any indexed file created with $CREATE/FDL can still be accessed by VAX PASCAL I/O statements. An indexed file can be created with the OPEN procedure only when the file components are of type RECORD. To create an indexed file with components of any other VAX PASCAL type, use $CREATE/FDL or a user action function in your program.

To define the key fields of an indexed file, you use the KEY attribute (see the *VAX PASCAL Reference Manual*). The KEY attribute can be applied only to fields of VAX PASCAL record types. If the record type has variants, the variant parts should not contain key fields.

One of the key fields, called the primary key, is identified as key number 0 and must be present in every file component when an indexed file is created. Additional keys, called alternate keys, can also be defined; they are numbered from 1 through a maximum of 254. While an indexed file can have as many as 255 key fields defined, in practice few applications require more than three or four key fields. By specifying the desired options with the KEY attribute, you can create an ascending or descending key. You can also specify if changes can be made to an alternate key and if duplicates of a key are allowed. See the *VAX PASCAL Reference Manual* for more information.

When you design an indexed file, you decide which byte positions within each component are the key fields. The key data types supported by VAX PASCAL are PACKED ARRAY OF CHAR and the ordinal types. If a key field of an ordinal type has a size attribute, it must be allocated as a signed word, a signed longword, an unsigned byte, an unsigned word, or an unsigned longword. Without a size attribute, the VAX PASCAL compiler will correctly allocate key fields by default.

The following sample program for a mail-order firm shows how you might define key fields in a file component:

```
TYPE
   Mail_Order : RECORD
                Order_Num : [KEY(0)] INTEGER;
                Name      : PACKED ARRAY[1..20] OF CHAR;
                Address   : PACKED ARRAY[1..20] OF CHAR;
                City      : PACKED ARRAY[1..19] OF CHAR;
                State     : PACKED ARRAY[1..2] OF CHAR;
                Zip_Code  : [KEY(1)] PACKED ARRAY[1..5] OF CHAR;
                Item_Num  : [KEY(2)] INTEGER;
                Shipping  : REAL;
                END;

VAR
   Order_File : FILE OF Mail_Order;
   Order_Rec  : Mail_Order;
```

Given this record type definition, you could use the following OPEN statement to create an indexed file:

```
OPEN (Order_File,
     'CUSTOMERS.DAT',
     HISTORY := NEW,
     ACCESS_METHOD := KEYED,
     ORGANIZATION  := INDEXED);
```

The type definition establishes three key fields, one primary key and two alternate keys, for the record type Mail_Order. The file Order_File that is opened by the OPEN procedure is declared as a file of Mail_Order records. The OPEN parameters define the file as an indexed file opened for keyed access.

The following RMS default KEY attribute options apply to the creation of an indexed file:

- Primary key values cannot be changed when a record is rewritten.
- Primary key values cannot be duplicated; that is, no two records can have the same primary key value.
- Alternate keys can both be changed and have duplicates.

You can override these defaults by specifying the desired option on the KEY attribute. See the *VAX PASCAL Reference Manual* for more information about the KEY attribute.

The *VAX Record Management Services Reference Manual* has more information on indexed file options. The $CREATE/FDL utility is explained in detail in the *VAX/VMS File Definition Language Facility Reference Manual*.

6.8.2 Using the Current Component and Next Component Pointers

RMS maintains two pointers into an open indexed file: the next component pointer and the current component pointer. When you reset an indexed file, the current component pointer indicates the file component with the lowest primary key value. Subsequent GET operations cause the current component pointer to indicate the component with the next higher key value in the same key field. If duplicate key values occur, the components are retrieved in the order in which they were written.

The current component is always the one most recently locked by a successful call to GET, RESET, RESETK, FIND, or FINDK; it is undefined until the file is either locked by one of these procedures or operated upon by any other file operation. The UPDATE and DELETE procedures (see Sections 6.8.5 and 6.8.6) operate on the current file component; thus, if one of these procedures is called while the current component is undefined or unlocked, an error results.

The next component pointer indicates the component to be retrieved by the next GET operation.

6.8.3 Writing Indexed Files

You can write components to an indexed file with either the WRITE or the PUT procedure. Each WRITE or PUT inserts a new component in the file and updates the indexes so that the new component appears in the correct order for each key field.

In the mail-order file example of Section 6.8.1, you could add a new component to the file with either of the following sets of statements.

```
WRITE (Order_File, Order_Rec);
    .
    .
    .
Order_File^ := Order_Rec;
PUT (Order_File);
```

It is possible to write two or more components with the same key value. Whether or not this duplicate key situation is allowed depends on the file characteristics specified when the file was created. By default, VAX PASCAL allows indexed files to have duplicate alternate keys but prohibits duplicate primary keys (see Section 6.8.1). Recall that you can override this option by using the DUPLICATE option on the KEY attribute. If duplicate keys are present in a file, the components with equal keys are retrieved in the order in which they were originally written.

For example, assume that five components are written to an indexed file in the following order (for clarity, only key fields are shown):

Order_Num	Zip_Code	Item_Num
1023	70856	375
942	02163	2736
903	14853	375
1348	44901	1047
1263	33032	690

If the file is later opened and read sequentially by primary key (Order_Num), then the sorted order of the components is unaffected by the duplicated Item_Num key:

Order_Num	Zip_Code	Item_Num
903	14853	375
942	02163	2736
1023	70856	375
1263	33032	690
1348	44901	1047

If the file is read along the second alternate key (Item_Num), however, the sort order is affected by the duplicate key.

Order_Num	Zip_Code	Item_Num
1023	70856	375
903	14853	375
1263	33032	690
1348	44901	1047
942	02163	2736

Notice that the components containing the same key value (375) were retrieved in the order in which they were written to the file.

6.8.4 Reading Indexed Files

You can read components of an indexed file with either the RESETK or the FINDK procedure. These procedures position the file pointers (see Section 6.8.2) at a particular component, determined by the key number, the key value, and the match type. Once you retrieve a component by key, you can use READ or GET statements to retrieve components with increasing or decreasing key values depending on the key's defined collating sequence.

Because the alternate key number 1 zip code defaults to an ascending collating sequence, the following example prints the order number and zip code of each file component that has a zip code greater than or equal to 10000 but less than 50000:

```
FINDK (Order_File, 1, '10000', NXTEQL);
Continue := TRUE;
WHILE Continue AND NOT UFB (Order_File) DO
   BEGIN
   READ (Order_File, Order_Rec);
   IF Order_Rec.Zip_Code < '50000'
   THEN
     WRITE (Report_File, 'Order number', Order_Rec.Order_Num,
           'has zip code', Order_Rec.Zip_Code)
   ELSE
      Continue := FALSE;
   END;
```

If you want to detect whether there is data available to be read from an indexed file, you can test the status of the file with the UFB function, as in the above example. If the value of UFB is TRUE, no record can be read; if UFB is FALSE, the READ procedure will successfully retrieve a record (see the *VAX PASCAL Reference Manual* for a full description of the UFB function).

6.8.5 Updating Components

To update existing components of an indexed file, use the UPDATE procedure. You cannot replace an existing component by writing it again; a WRITE or PUT procedure attempts to add a new component.

An update operation is accomplished in two steps. First, you must lock the component, thereby making it the current component. Next, you must execute the UPDATE procedure. For example, to update the component containing Order_Num 903 (see prior examples) so that the Name field becomes 'Theodore Zinck', you might use the following statements:

```
FINDK (Order_File, 0, 903, EQL);
Order_File^.Name := 'Theodore Zinck';
UPDATE (Order_File);
```

When you update a component, RMS defaults allow you to change the values of any key fields except the primary key. You can change the RMS defaults by specifying the NOCHANGES option on the KEY attribute to prohibit the changing of alternate key values.

The following example shows the updating of a file to add shipping charges to all orders mailed to areas with zip codes less than 50000:

```
RESETK (Order_File, 1);
Continue := TRUE;
WHILE Continue AND NOT UFB (Order_File) DO
   IF Order_File^.Zip_Code < '50000'
   THEN
      BEGIN
      Order_File^.Shipping := 1.00;
      UPDATE (Order_File);
      GET (Order_File);
      END
   ELSE
      Continue := FALSE;
```

This example uses the UPDATE procedure to change the value in the field Order_File^.Shipping. Because components must be locked before the update can occur, the GET procedure is used here to retrieve the next component. GET locks the retrieved component; the READ procedure does not.

6.8.6 Deleting Components

To delete components from an indexed file, use the DELETE procedure. The DELETE procedure, like the UPDATE procedure, requires that the component be locked before the procedure executes.

The following example deletes the second component in the file with Item_Num equal to 375 (see prior examples):

```
FINDK (Order_File, 2, 375, EQL);
GET (Order_File);
IF (Order_File^.Item_Num = 375)
THEN
   DELETE (Order_File)
ELSE
   WRITELN ('There is no second component');
```

Deletion removes a component from all defined indexes in the file.

6.8.7 Recovering from Exception Conditions

When using indexed files, you can expect to encounter certain exception conditions. Exception conditions usually result from operations such as the following:

- An attempt to read a file component that does not exist
- An attempt to write a record that invalidly duplicates a key
- An attempt to read a locked file component

If the component specified by a FINDK procedure does not exist, an error does not occur; the value of UFB simply becomes TRUE. Your program should test the status of UFB for this possibility.

Other exception conditions can be handled by the STATUS function and by the error recovery parameter that is available with I/O procedures. The STATUS function indicates the status of a file following the last operation performed on it. The error recovery parameter indicates the action to be taken should an I/O procedure fail. You can also write condition handlers (see Chapter 7) to allow your program to recover from I/O errors, although this method is not the simplest.

The following example shows how the use of the STATUS function and the error recovery parameter allow a program to recover from an exception condition:

```
READ (Flight, No_of_Passengers);
1:  FINDK (Reservation_File, 0, Flight, EQL, ERROR := CONTINUE);
IF STATUS (Reservation_File) = PAS$K_FAIGETLOC
THEN
   BEGIN
      WRITE ('Reservation record locked.  Try again',
         '(Yes or No) : ');
      READ (Response);
      If Response = Yes
      THEN
         GOTO 1;
   END

ELSE
   IF STATUS (Reservation_File) = 0
   THEN
      BEGIN
         Reservation_File^.Passenger_Count :=
            Reservation_File^.Passenger_Count + No_of_Passengers;
         UPDATE (Reservation_File);
      END
   ELSE
      WRITE ('Error accessing Reservation_File');
```

In this example, PAS$K_FAIGETLOC is the symbolic name that represents the record-locked error code. Following the FINDK procedure, STATUS tests whether the specified component of Reservation_File was locked. Because the ERROR parameter specifies that execution should continue in the event of an error, the FINDK procedure is repeatedly attempted until the record is no longer locked. When STATUS returns a value of 0, indicating that FINDK was successful, the passenger count field is increased. When STATUS returns a value other than 0 or the value of PAS$K_FAIGETLOC, the program prints an error message.

6.9 Input/Output Considerations for Text Files

The RMS unit of transfer is a file structure called a record, which should not be confused with the Pascal type RECORD. For all VAX PASCAL file types other than text files, there is a one-to-one correspondence between a VAX PASCAL file component and an RMS record. For example, each RMS record that constitutes a FILE OF INTEGER type contains the representation of one integer value. A component of a text file is a single character; however, an RMS record in a text file is equivalent to a complete line of characters as terminated by an end-of-line marker.

The processing of input and output for text files in VAX PASCAL requires special considerations, as described in the following sections.

6.9.1 Text File Buffering

RMS always deals with complete records. When characters are being transferred between a physical I/O device and a VAX PASCAL text file, the record is not complete until an end-of-line marker is detected. Therefore, as the characters are read or written, they are stored in an internal line buffer until a complete record can be transferred through RMS. The line buffer contains characters of a partially processed line of text.

Because RMS records correspond exactly to the components of nontext files, no buffering is necessary for files of any type other than TEXT.

6.9.2 FILE OF CHAR File Type

The file type FILE OF CHAR differs from the predefined file type TEXT in that FILE OF CHAR allows a single character to be the unit of transfer between a program and its associated I/O devices. Single-character RMS records are always read with the READ procedure, and must be read exclusively into variables of type CHAR, including CHAR components of structured variables. The EOLN, READLN, and WRITELN routines cannot be used on files of type FILE OF CHAR because such files, unlike text files, do not have end-of-line markers.

6.9.3 Prompting of Text Files

A prompt is a line of text that appears on the terminal screen when a partial RMS record is written to a terminal output file. Normally, when a WRITE procedure is performed on a text file connected to a terminal, the individual characters are accumulated in the line buffer until a subsequent WRITELN procedure is executed. In effect, WRITELN generates an end-of-line marker and thereby completes an RMS record. When the record is completed or the file is closed, a full line of characters is written to the specified text file.

The VAX PASCAL run-time system is capable of dealing with partial records, however, when characters are being written to a terminal output file opened with the LIST carriage control option (LIST is the default). Partial records are written to the terminal before input is transferred from any terminal to the line buffer of a text file. In this situation, VAX PASCAL searches for all text files opened for output on terminals; it then writes to those files any partial records contained in the files' respective line buffers. These partial records, called prompts, appear on the terminal screens. You respond to a prompt by typing a line of input data terminated by a break character. Usually a line of input from the terminal ends with a carriage return, but you can type any VAX/VMS break character (including ESCAPE) to mark the end of a line.

You will also be prompted for input when your program tests for either end-of-line (with the EOLN function) or end-of-file (with the EOF function), and the internal line buffer of the tested file is empty. To perform the test, partial lines of output are transferred from each line buffer to the corresponding terminals. The file's line buffer is filled by the input you type in response to the prompt. If you immediately press CTRL/Z, an end-of-file condition exists on the terminal and EOF will return TRUE. If you type any other VAX/VMS break character, the internal line buffer will contain an end-of-line marker and EOLN will return TRUE. Any other input you type will be placed in the internal line buffer.

Prompting does not occur on terminal output files that have a carriage control option other than LIST or that were opened by a user action function.

6.9.4 Delayed Device Access to Text Files

The Pascal standard specifies that the file buffer must always contain the next file component that the program will process. This requirement can cause problems when the input to the program depends on the output most recently generated. To alleviate these problems in the processing of text files, VAX PASCAL uses a technique called delayed device access, also known as "lazy lookahead."

As a result of delayed device access, input is not retrieved from a physical file device and inserted in the file buffer until the program is ready to process it. The file buffer is filled when the program makes the next reference to the file. A reference to the file consists of any use of the file buffer variable, including its implicit use in the GET, READ, and READLN procedures, or any explicit test for the status of the file, namely, the EOF, EOLN, STATUS, and UFB functions.

The RESET procedure, which is required when any text file is opened for input, initiates the process of delayed device access. (Note that RESET is done automatically on the predeclared file INPUT.) RESET expects to fill the file buffer with the first component of the file. However, because of delayed device access, data is not supplied from the input device to fill the file buffer until the next reference to the file occurs.

When writing a program in which a text file will supply the input, you should be aware that delayed device access will occur. Because RESET initiates delayed device access and EOF and EOLN cause the file buffer to be filled, you should place the first prompt for input before any tests for EOF or EOLN. The data you enter in response to the prompt is retained by the file device until you make another reference to the input file.

The following example illustrates the use of prompts in the reading of input data and delayed device access:

```
VAR
   Purch_Amount : REAL;
     .
     .
     .
WRITE ('Enter amount of purchase or <CTRL/Z>: ');
WHILE NOT EOF DO
   BEGIN
   READLN (Purch_Amount);
   WRITE ('Enter amount of purchase or <CTRL/Z>: ');
   END;
```

The first reference to the file INPUT is the EOF test in the WHILE statement. When the test is performed, the VAX PASCAL run-time system attempts to read a line of input from the text file. Therefore, in this program, it is important to prompt for the amount of purchase before testing for EOF. If you respond to the prompt by pressing CTRL/Z, EOF returns TRUE. If you respond by entering a purchase amount, EOF returns FALSE.

If you respond to the first prompt for input by typing a real number, access to the input device is delayed until the EOF function makes the first reference to the file INPUT. The EOF function causes a line of text to be read into the internal line buffer. The subsequent READLN procedure reads the input value from the line of text and assigns it to the variable Purch_Amount. The final statement in the WHILE loop is the request for another input value. The WHILE loop is executed until EOF detects the end-of-file marker.

A sample run of a program containing this loop might be as follows:

```
$ RUN PURCH
Enter amount of purchase or <CTRL/Z>: 7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.99
Enter amount of purchase or <CTRL/Z>: CTRL/Z
$
```

The following program fragment illustrates a method of writing the same loop that does not take into account delayed device access and therefore produces incorrect results:

```
WHILE NOT EOF DO
   BEGIN
   WRITE ('Enter amount of purchase or <CTRL/Z>: ');
   READLN (Purch_Amount);
   END;
```

The EOF test at the beginning of the loop causes the file buffer to be filled. However, because no input has yet been supplied, the prompt will not appear on the terminal screen until you have supplied input to fill the INPUT file buffer.

A sample run of a program containing this loop might be as follows:

```
$ RUN PURCHASE
7.95
Enter amount of purchase or <CTRL/Z>: 6.49
Enter amount of purchase or <CTRL/Z>: 19.95
Enter amount of purchase or <CTRL/Z>: CTRL/Z
$
```

Note that the prompt always appears *after* you have typed a value for Purch_Amount.

6.9.5 Writing Partial Lines to Terminals

The WRITE procedure buffers output to the terminal until the WRITELN procedure is called. If too many characters are buffered, it can cause the VAX PASCAL Run-Time Library internal buffer to overflow. The default size for this buffer is 133 characters (255 characters if you are using VAX/VMS Version 4.6 or higher) for text files. If you want to increase the internal buffer size, you can explicitly open the predeclared file OUTPUT with a larger record length. For example:

```
OPEN(OUTPUT,RECORD_LENGTH := 512);
```

If you want each WRITE procedure to output directly to the terminal without buffering until the next WRITELN, you can explicitly open the predeclared file variable OUTPUT without carriage control. In this mode, the WRITELN procedure will write the information to the file without adding any carriage control. However, you need to include the carriage return and the line feed characters in the output strings; the WRITELN procedure no longer provides these automatically. For example:

```
OPEN(OUTPUT, CARRIAGE_CONTROL := NONE);

WRITELN(''(10)'Output this');
WRITELN('string directly');
WRITELN('to the terminal'(13));
```

This is useful when you are writing escape sequences or other graphics characters to terminal devices.

6.10 Interprocess Communication (Mailboxes)

The exchange of data between processes is often useful; for example, such an exchange can synchronize execution or 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$CREMBX) system service. See the *VAX/VMS System Services Reference Manual* for information about using SYS$CREMBX.

Data transmission by means of mailboxes is synchronous; a program that uses a mailbox to read information from another process must wait until that process has sent the information to the mailbox. Likewise, a program that writes information into a mailbox must wait until the other process has read the information before execution can continue. When two processes exchange information between text files, delayed device access occurs as it always does for text files. When information is exchanged between nontext files, file components are not buffered; as soon as one process writes a component, the other process can read it.

When a process closes a mailbox file that it opened for writing (that is, the mailbox was opened without HISTORY:= READONLY), an end-of-file message is written to the mailbox. The next process to read from the mailbox will receive an end-of-file condition. If you plan only to read from a mailbox, you should specify HISTORY := READONLY when you open it to prevent unwanted end-of-file conditions when the mailbox is closed.

The following program will write to a mailbox with the logical name FOZZIE_BEAR. Because $QIOW is used and no function code modifier is present, the write will not complete until the program MAILR reads the mailbox.

MAILS.PAS—write to mailbox

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Mails (OUTPUT);

CONST
    Mailbox_Name = 'fozzie_bear';
    Message      = 'this is a message';

TYPE
    Uword    = [WORD] 0..65535;
    IO_Block = RECORD
        IO_Stat, Count : Uword;
        Dev_Info       : INTEGER;
        END;

VAR
    Channel  : Uword;
    Sys_Stat : INTEGER;
    Stat_Blk : IO_Block;

PROCEDURE LIB$STOP( Cond_Value : [IMMEDIATE] INTEGER ); EXTERN;

BEGIN
Sys_Stat := $CREMBX( Chan := Channel, Lognam := Mailbox_Name );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
```

```
Sys_Stat := $QIOW( Chan := Channel,
                   Func := IO$_WRITEVBLK,
                   Iosb := Stat_Blk,
                   P1   := Message,
                   P2   := LENGTH(Message) );

IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
IF NOT ODD( Stat_Blk.IO_Stat ) THEN LIB$STOP( Stat_Blk.IO_Stat );

WRITELN('MAILS has completed mailbox write.');
END.
```

The following program will read a message written to a mailbox by program MAILS. As soon as it reads the message, MAILS will continue.

MAILR.PAS—read from mailbox

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Mailr (OUTPUT);

CONST
    Mailbox_Name = 'fozzie_bear';

TYPE
    Uword    = [WORD] 0..65535;
    IO_Blk   = RECORD
        IO_Stat, Count : Uword;
        Dev_Info       : INTEGER;
        END;

VAR
    Channel    : Uword;
    Read_Input : VARYING [30] OF CHAR;
    IO_Statblk : IO_Blk;
    Sys_Stat   : INTEGER;

PROCEDURE LIB$STOP( Cond_Value : [IMMEDIATE] INTEGER ); EXTERN;

BEGIN
Sys_Stat := $CREMBX( Chan := Channel, Lognam := Mailbox_Name );
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );

Sys_Stat := $QIOW( Chan := Channel,
                   Func := IO$_READVBLK,
                   Iosb := IO_Statblk,
                   P1   := Read_Input.Body,
                   P2   := SIZE(Read_Input.Body));
IF NOT ODD( Sys_Stat ) THEN LIB$STOP( Sys_Stat );
IF NOT ODD( IO_Statblk.IO_Stat ) THEN LIB$STOP( IO_Statblk.IO_Stat );

Read_Input.Length := IO_Statblk.Count;

WRITELN( Read_Input );
END.
```

Because both programs, MAILS and MAILR, use $CREMBX to create a temporary mailbox with the logical name FOZZIE_BEAR if no such mailbox exits, and then assign a channel to it, the programs can be run in any order. For example:

```
$! From one process
$ RUN MAILS
  MAILS has completed mailbox write

$! From a subprocess
$ RUN MAILR
  This is a message
```

6.11 Communication with Remote Computers (Networks)

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

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, you must specify a node name as the first element of a file specification. For example:

```
BOSTON::DISK$USER:[SMITH]TEST.DAT;21
```

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, enclosing this information in quotation marks ("). For example:

```
BOSTON::"TASK=SLAVE"
```

The example specifies a task using the command file SLAVE.COM on the BOSTON node of the network.

The following example shows how messages can be exchanged between local and remote processes by means of VAX PASCAL I/O procedures.

MASTER.PAS—local process

```
PROGRAM MASTER(INPUT,OUTPUT,Net_Link);

VAR
    Net_Link : TEXT;
    Message  : INTEGER;

    BEGIN

    { Open a DECnet link with another process to exchange messages }
    OPEN(
        File_Variable := Net_Link,
        File_Name     := 'BOSTON"ACCOUNT PASSWORD"::"task=slave"',
        History       := OLD);

    WRITE('Input a message number to send:');
    REWRITE(Net_Link);

    WHILE NOT EOF DO
        BEGIN

        { Input the number and send it down the link }
        READLN(message);
        WRITELN(Net_Link,Message);

        { Reverse the link and wait for the reply }
        RESET(Net_Link);
        READLN(Net_Link,Message);
        WRITELN('Received message = ',Message);

        { Reverse the link and send another message }
        REWRITE(Net_Link);
        WRITE('Input a message number to send:');
        END;

    CLOSE(Net_Link);
    END.
```

SLAVE.PAS—remote process

```
PROGRAM Slave(INPUT,OUTPUT,Net_Link);

VAR
    Net_Link : TEXT;
    Message : INTEGER;

    BEGIN

    { Establish the logical link from the master }
    OPEN(
        File_Variable := Net_Link,
        File_Name     := 'SYS$NET',
        History       := OLD);

    { Prepare to read messages }
    RESET(Net_Link);
    WHILE NOT EOF(Net_link) Do
        BEGIN

        { Read the message sent from the master }
        READLN(Net_Link,Message);

        { Reverse the link and send back the reply }
        REWRITE(Net_Link);
        WRITELN(Net_Link,Message);

        { Reverse the link to prepare to read again }
        RESET(Net_Link);
        END;

    CLOSE(Net_Link);
    END.
```

SLAVE.COM

```
$! Invoke SLAVE.EXE to complete the logical link.
$!
$ RUN SLAVE
```

Because the master program opens the DECnet link using the special notation TASK=SLAVE, you can establish the link by executing the command file SLAVE.COM from the local BOSTON node.

The following example shows how you can write a remote file using VAX PASCAL I/O procedures:

```
PROGRAM UPDATE (Newdat, Branch);

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


BEGIN           (* main program *)
RESET (Newdat);
OPEN (Branch,
      'NASHUA"PLUGH XYZZY"::MASTER.DAT',
      HISTORY := NEW);
REWRITE (Branch);

WHILE NOT EOF (Newdat) DO
   BEGIN
   Branch^ := Newdat^;
   GET (Newdat);
   PUT (Branch);
   END;
CLOSE (Branch);
CLOSE (Newdat);
END.            (* main program *)
```

This example 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 default device and directory of user PLUGH with password XYZZY.

If you use logical names in your program, you can equate them 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::DISK$APPLIC_SET:file-name  LOGIC
```

Local Access

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

The MOUNT and ASSIGN commands are described in detail in the *VAX/VMS DCL Dictionary.*

DECnet capabilities are described in the *Guide to Networking on VAX/VMS.*

Chapter 7

Error Processing and Condition Handling

During the execution of a VAX PASCAL program, errors and exception conditions can occur. They can result from errors during I/O operations, invalid input data, incorrect calls to library routines, arithmetic errors, or system-detected errors. VAX PASCAL provides two methods of error control and recovery:

- VAX Run-Time Library default error-processing procedures
- The VAX Condition Handling Facility (including user-written condition handlers)

These error-processing methods are complementary and can be used in the same program.

The VAX Run-Time Library can provide all the condition handling a program needs. By default, it generates error messages for all errors and exception conditions that occur during program execution. Section 7.1 describes how the Run-Time Library processes VAX PASCAL errors.

The VAX Condition Handling Facility provides, at the lowest level, all condition handling for the Run-Time Library. It can also accommodate user-written condition handlers for processing conditions that occur during program execution. Sections 7.2 through 7.5 present a general discussion of condition handling, describe how to write a condition handler in VAX PASCAL, and show how to handle faults and traps. Examples of condition handling are given in Section 7.6.

The use of condition handlers requires considerable programming experience and should not be undertaken by novice users. You should understand the discussions of condition handling in the *VAX/VMS Run-Time Library Routines Reference Manual*, the *VAX/VMS System Services*

Reference Manual, and the *Introduction to VAX/VMS System Routines* before attempting to write your own condition handler.

7.1 Run-Time Library Default Error Processing

When a run-time error occurs, the Run-Time Library prints a message and terminates program execution. These default actions happen unless your program includes a condition handler.

Run-time errors are reported in the following format:

```
%PAS-L-ident, message-text
```

The L portion is a 1-letter abbreviation indicating the severity of the error (see Section 7.4.4). The ident portion is an abbreviation of the error message described in the message text. VAX PASCAL run-time errors and recovery procedures are described in Appendix A. Most Run-Time Library routines provide their own error messages, as described in the *VAX/VMS Run-Time Library Routines Reference Manual.*

7.2 Definitions of Terms

The following terms are used in the discussion of condition handling:

- *Condition handler*—A function specified by a routine as the handler to be called when an exception condition is signaled.
- *Condition value*—An integer value that identifies a specific condition.
- *Stack frame*—A standard data structure built on the stack during a routine call, starting from the location addressed by the frame pointer (FP) and proceeding to both higher and lower addresses; it is popped off the stack during the return from a routine.
- *Routine activation*—The environment in which a routine executes. This environment includes a unique stack frame on the run-time stack; the stack frame contains the address of a condition handler for the routine activation. A new routine activation is created every time a routine is called and is deleted when control passes from the routine.
- *Establish*—The process of placing the address of a condition handler in the stack frame of the current routine activation. A condition handler established for a routine activation is automatically called when a condition occurs. In VAX PASCAL, condition handlers are established

by means of the predeclared procedure ESTABLISH. A routine that establishes a condition handler is known as an *establisher.*

- *Program exit status*—The status of the program at its completion.
- *Signal*—The means by which the occurrence of an exception condition is made known. Signals are generated by the operating system in response to I/O events and hardware errors, by the system-supplied library routines, and by user routines. All signals are initiated by a call to the signaling facility, for which there are two entry points:
  - LIB$SIGNAL—Used to signal a condition and, possibly, to continue program execution
  - LIB$STOP—Used to signal a severe error and discontinue program execution, unless a condition handler performs an unwind operation
- *Resignal*—The means by which a condition handler indicates that the signaling facility is to continue searching for a condition handler to process a previously signaled error. To resignal, a condition handler returns the value SS$_RESIGNAL.
- *Unwind*—The return of control to a particular routine activation, bypassing any intermediate routine activations. For example, if X calls Y, and Y calls Z, and Z detects an error, then a condition handler associated with X or Y can unwind to X, bypassing Y. Control returns to X immediately following the point at which X called Y.

7.3 Overview of Condition Handling

When the VAX/VMS system creates a user process, a system-defined condition handler is established in the absence of any user-written condition handler. The system-defined handler processes errors that occur during execution of the user image. Thus, by default, a run-time error causes the system-defined condition handler to print error messages and to terminate or continue execution of the image, depending on the severity of the error.

When a condition is signaled, the system searches for condition handlers to process the condition. The system conducts the search for condition handlers by proceeding down the stack, frame by frame, until a condition handler is found that does not resignal. The default handler calls the system's message output routine to send the appropriate message to the user. Messages are sent to the SYS$OUTPUT and SYS$ERROR files. If the condition is not a severe error, program execution continues. If the condition is a severe error, the default handler forces program termination, and the condition value becomes the program exit status.

You can create and establish your own condition handlers according to the needs of your application. For example, a condition handler could create and display messages that describe specific conditions encountered during the execution of your program, instead of relying on system error messages.

7.3.1 Condition Signals

A condition signal consists of a call to either LIB$SIGNAL or LIB$STOP, the two entry points to the signaling facility. These entry points must be declared as external procedures. For example:

```
[EXTERNAL,ASYNCHRONOUS]
PROCEDURE LIB$SIGNAL (
   Condition : [IMMEDIATE] INTEGER;
   FAO_Params : [UNSAFE,LIST,IMMEDIATE] INTEGER);
   EXTERN;

[EXTERNAL,ASYNCHRONOUS]
PROCEDURE LIB$STOP (
   Condition : [IMMEDIATE] INTEGER;
   FAO_Params : [UNSAFE,LIST,IMMEDIATE] INTEGER);
   EXTERN;
```

Additional parameters can be included to provide supplementary information about the condition.

If a condition occurs in a routine that is not prepared to handle it, a signal is issued to notify other active routines. If the current routine can continue after the signal is propagated, you can call LIB$SIGNAL. A higher-level routine can then determine whether program execution should continue. If the nature of the condition does not allow the current routine to continue, you can call LIB$STOP.

7.3.2 Handler Responses

A condition handler responds to an exception condition by taking action in three major areas:

- Condition correction
- Condition reporting
- Execution control

The handler first determines whether the condition can be corrected. If so, it takes the appropriate action, and execution continues. If the handler cannot correct the condition, the condition may be resignaled; that is, the handler requests that another condition handler be sought to process the condition.

A handler's condition reporting 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 the output file
- Changing the severity field of the condition value and resignaling the condition
- Signaling a different condition, for example, the production of a message designed for a specific application

A handler can control execution in several ways:

- By continuing from the signal. If the signal was issued through a call to LIB$STOP, the program exits.
- By doing a nonlocal GOTO operation (see Section 7.6, Example 5).
- By unwinding to the establisher at the point of the call that resulted in the exception. The handler can then determine the function value returned by the called routine.
- By unwinding to the establisher's caller (the routine that called the routine which established the handler). The handler can then determine the function value returned by the called routine.

7.4 Writing Condition Handlers

The following sections describe how to write and establish condition handlers and provide some simple examples. See the *Introduction to VAX/VMS System Routines*, the *VAX/VMS Run-Time Library Routines Reference Manual*, and the *VAX/VMS System Services Reference Manual* for more details on condition handlers.

7.4.1 Establishing and Removing Handlers

To use a condition handler, you must first declare the handler as a routine in the declaration section of your program; then, within the executable section, you must call the predeclared procedure ESTABLISH. The ESTABLISH procedure sets up a VAX PASCAL language-specific condition handler that in turn allows your handler to be called. User-written condition handlers set up by ESTABLISH must have the ASYNCHRONOUS attribute and two integer array formal parameters. Such routines can access only local, read-only, and volatile variables, and local, predeclared, and asynchronous routines.

Because condition handlers are asynchronous, any attempt to access a non-read-only or nonvolatile variable declared in an enclosing block will result in a warning message. The predeclared file variables INPUT and OUTPUT are such nonvolatile variables; therefore, simultaneous access to these files from both an ordinary program and from an asynchronous condition handler's activation may have undefined results. The following steps outline the recommended method for performing I/O operations from a condition handler:

1. Declare a file with the VOLATILE attribute at program level.
2. Open this file to refer to SYS$INPUT, SYS$OUTPUT, or another appropriate file.
3. Use this file in the condition handler.

External routines (including system services) that are called by a condition handler require the ASYNCHRONOUS attribute in their declaration.

You should set up a user-written condition handler with the predeclared procedure ESTABLISH rather than with the Run-Time Library routine LIB$ESTABLISH. ESTABLISH follows the VAX PASCAL procedure-calling rules and is able to handle VAX PASCAL condition handlers more efficiently than LIB$ESTABLISH. A condition handler set up by LIB$ESTABLISH might interfere with the default error handling of the VAX PASCAL run-time system, and cause unpredictable results.

To establish a condition handler with either LIB$ESTABLISH or ESTABLISH, your routine must have the ASYNCHRONOUS attribute and two integer array formal parameters. It can access only static, volatile, or read-only variables, asynchronous routines declared at the outermost level of the program, and predeclared VAX PASCAL routines.

The following example shows how to establish a condition handler using the VAX PASCAL procedure ESTABLISH:

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr)    : INTEGER;
   EXTERN;
   .
   .
   .
ESTABLISH (Handler);
```

To establish the handler, call the ESTABLISH procedure. To remove an established handler, call the predeclared procedure REVERT, as follows:

```
REVERT;
```

As a result of this call, the condition handler established in the current stack frame is removed. When control passes from a routine, any condition handler established during the routine's activation is automatically removed.

7.4.2 Declaring Parameters for Condition Handlers

A VAX PASCAL condition handler is an integer-valued function that is called when a condition is signaled. Two formal VAR parameters must be declared for a condition handler:

- An integer array to refer to the parameter list from the call to the signal routine (the signal array); that is, the list of parameters included in calls to LIB$SIGNAL or LIB$STOP (see Section 7.3.1)
- An integer array to refer to information concerning the routine activation that established the condition handler (the mechanism array)

For example, a condition handler can be defined as follows:

```
TYPE
   Sigarr  = ARRAY[0..9] OF INTEGER;
   Mecharr = ARRAY[0..4] OF INTEGER;
```

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION Handler
   (VAR Sigargs : Sigarr;
    VAR Mechargs : Mecharr)    : INTEGER;
   EXTERN;
       .
       .
       .
ESTABLISH (Handler);
       .
       .
       .
```

The signal procedure passes the values listed below to the array Sigargs.

Value	Description
Sigargs[0]	The number of parameters being passed in this array (parameter count).
Sigargs[1]	The primary condition being signaled (condition value). See Section 7.4.4 for a discussion of condition values.
Sigargs[2 to n]	The optional parameters supplied in the call to LIB$SIGNAL or LIB$STOP; note that the index range of Sigargs should include as many entries as are needed to refer to the optional parameters.

The routine that established the condition handler passes the values listed below to the array Mechargs. These values contain information about the establisher's routine activation.

Value	Description
Mechargs[0]	The number of parameters being passed in this array
Mechargs[1]	The address of the stack frame that established the handler
Mechargs[2]	The number of calls that have been made (that is, the stack frame depth) from the routine activation up to the point at which the condition was signaled
Mechargs[3]	The value of register R0 at the time of the signal
Mechargs[4]	The value of register R1 at the time of the signal

7.4.3 Handler Function Return Values

Condition handlers are functions that return values to control subsequent execution. These values and their effects are listed below.

Value	Effect
SS$_CONTINUE	Continues execution from the signal. If the signal was issued by a call to LIB$STOP, the program does not continue, but 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:

```
[ASYNCHRONOUS,EXTERNAL(SYS$UNWIND)] FUNCTION $UNWIND
   (DEPADR : INTEGER := %IMMED 0;
   NEWPC   : INTEGER := %IMMED 0)     : INTEGER;
   EXTERNAL;
```

When SYS$UNWIND is called, the function return value of the condition handler is ignored. The handler modifies the saved registers R0 and R1 in the mechanism parameters to specify the called function's return value.

A stack unwind is usually made to one of two places:

- The point in the establisher at which the call was made that resulted in the exception. Specify the following:

  ```
  Status := $UNWIND (Mechargs[2],0);
  ```

- The routine that called the establisher. Specify the following:

  ```
  Status := $UNWIND (Mechargs[2]+1,0);
  ```

7.4.4 Condition Values and Symbols

The VAX/VMS system uses condition values to indicate that a called routine has either executed successfully or failed, and to report exception conditions. Condition values are usually symbolic names that represent 32-bit packed records, consisting of fields (usually interpreted as integers) that indicate which system component generated the value, the reason the value was generated, and the severity of the condition. The definition of a condition value can have the following form:

```
TYPE Condition_Value = PACKED RECORD

(* Field                           Bits        Meaning *)

Severity:[POS(0),BIT(3)] 0..7;     (* 0..2     One of the following severity
                                               codes:
                                               0 - warning
                                               1 - success
                                               2 - error
                                               3 - information
                                               4 - severe error
                                               5,6,7 - reserved *)

Message:[POS(3),BIT(13)]0..8191;   (* 3..15    The condition that occurred.
                                               When bit 15 is 1, it indicates
                                               that the message is specific
                                               to a single facility. When
                                               bit 15 is 0, it indicates a
                                               system-wide message. *)

Facility:[POS(16),BIT(12)]0..4095; (*16..27    The software component that
                                               generated the condition
                                               value. When bit 27 is 1, it
                                               indicates a customer facility.
                                               When bit 27 is 0, it indicates
                                               a DIGITAL facility. *)

Control:[POS(28),BIT(4)]0..15;     (*28..31    Control bits. *)

END;
```

A warning severity code (0) indicates that although output was produced, the results may be unpredictable. An error severity code (2) indicates that output was produced even though an error was detected. Execution can continue, but the results may not be correct. A severe error code (4) indicates that the error was of such severity that no output was 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.

Occasionally a condition handler may require a particular condition to be identified by an exact match; that is, each bit of the condition value bits (0..31) must match the specified condition. For example, you may want to process a floating overflow condition only if the severity code is still 4 (that is, if no previous condition handler has changed the severity code) and the control bits have not been modified. A typical condition handler response is to change the severity code and resignal.

In most cases, however, you want some response to a condition, regardless of the value of the severity code or control bits. To ignore the severity and control fields of a condition value, declare and call the LIB$MATCH_COND function (for an example, see the declaration section in Section 7.6).

7.5 Fault and Trap Handling

If a VAX processor detects an error while executing a machine instruction, it can take one of two actions. The first action, called a fault, preserves the contents of registers and memory in a consistent state so that the instruction can be restarted. The second action, called a trap, completes the instruction, but with a predefined result. For example, if an integer overflow trap occurs, the result is the correct low-order part of the true value.

The action taken when an exception occurs depends on the type of exception. For example, faults occur for access violations and for detection of a floating reserved operand. Traps occur for integer overflow and for integer divide-by-zero exceptions. However, when a floating overflow, floating underflow, or floating divide-by-zero exception occurs, the action taken depends on the type of VAX processor executing the instruction. The original VAX-11/780 processor traps when these errors occur and stores a floating reserved operand in the destination. All other VAX processors fault on these exceptions, allowing the error to be corrected and the instruction restarted.

If your program is written to handle floating traps, but runs on a VAX processor that generates faults, execution may continue incorrectly. For example, if a condition handler merely causes execution to continue after a floating trap, a reserved operand is stored and the next instruction is executed. However, the same handler used on a processor that generates faults causes an infinite loop of faults because it restarts the erroneous instruction. Therefore, you should write floating-point exception handlers that take the appropriate actions for both faults and traps.

Separate sets of condition values are signaled by the processor for faults and traps. Exceptions and their condition code names are as follows:

Exception	Fault	Trap
Floating overflow	SS$_FLTOVF_F	SS$_FLTOVF
Floating underflow	SS$_FLTUND_F	SS$_FLTUND
Floating divide-by-zero	SS$_FLTDIV_F	SS$_FLTDIV

To convert faults to traps, you can use the Run-Time Library LIB$SIM_TRAP procedure either as a condition handler or as a called routine from a user-written handler. When LIB$SIM_TRAP recognizes a floating fault, it simulates the instruction completion as if a floating trap had occurred. For information on how to use this procedure, see the *VAX/VMS Run-Time Library Routines Reference Manual.*

7.6 Examples of Condition Handlers

The following declarations are used in the examples in this section.

```
[ASYNCHRONOUS,EXTERNAL(SYS$UNWIND)] FUNCTION $UNWIND
   (DEPADR : INTEGER := %IMMED 0;
   NEWPC   : INTEGER := %IMMED 0)     : INTEGER;
   EXTERNAL;

[EXTERNAL,ASYNCHRONOUS] FUNCTION LIB$MATCH_COND
   (Cond_Value : [UNSAFE] INTEGER;
   Cond_Vals   : [UNSAFE, LIST] INTEGER)   : INTEGER;
   EXTERNAL;

TYPE
   Sig_Args  = ARRAY[0..100] OF INTEGER;        {Signal parameters}
   Mech_Args = ARRAY[0..4] OF [UNSAFE] INTEGER; {Mechanism parameters}
   Cond_Status = [VOLATILE] RECORD              {Condition values}
        CASE INTEGER OF
             0:(STS$_SUCCESS   : [POS(0),BIT(1)] BOOLEAN);
             1:(STS$_SEVERITY  : [POS(0),BIT(3)] 0..2**3-1);
             2:(STS$_CODE      : [POS(3),BIT(12)] 0..2**12-1);
             3:(STS$_FAC_SP    : [POS(15),BIT(1)] BOOLEAN);
             4:(STS$_MSG_NO    : [POS(3),BIT(13)] 0..2**13-1);
             5:(STS$_CUST_DEF  : [POS(27),BIT(1)] BOOLEAN);
             6:(SYS$_FAC_NO    : [POS(16),BIT(12)] 0..2**12-1);
             7:(STS$_COND_ID   : [POS(3),BIT(25)] 0..2**25-1);
             8:(STS$_INHIB_MSG : [POS(28),BIT(1)] BOOLEAN);
             END;
```

Example 1

```
[ASYNCHRONOUS] FUNCTION Handler_0
   (VAR SA : Sig_Args;
   VAR MA : Mech_Args)     : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                      { do something appropriate }
          .
      Handler_0 := SS$_CONTINUE;  { condition handled,
                                    propagate no further }
      END
   ELSE
      Handler_0 := SS$_RESIGNAL;  { propagate condition
                                    status to other handlers }
   END;
```

This example shows a simple condition handler. The handler identifies the condition being signaled as one that it is prepared to handle and then takes appropriate action. Note that for all unidentified condition statuses, the handler resignals. A handler must always follow this behavior.

Example 2

```
[ASYNCHRONOUS] FUNCTION Handler_1
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)     : [UNSAFE] INTEGER;

   BEGIN
   IF SA[1] = SS$_UNWIND
   THEN
      BEGIN
          .
          .                         { cleanup }
          .
      END;
   Handler_1 := SS$_RESIGNAL;
   END;
```

When writing a handler, remember that it can be activated with a condition of SS$_UNWIND, signifying that the establisher's stack frame is about to be unwound. If the establisher has special cleanup to perform, such as freeing dynamic memory, closing files, or releasing locks, the handler should check for the SS$_UNWIND condition status. If there is no cleanup, the required action of resignaling all unidentified conditions results in the correct behavior. On return from a handler activated with SS$_UNWIND, the stack frame of the routine that established the handler is deleted (unwound).

Example 3

```
[ASYNCHRONOUS] FUNCTION Handler_2
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      BEGIN
          .
          .                        { cleanup }
          .
      MA[3] := expression;         { establish function result
                                     seen by caller }
     $UNWIND;                      { unwind to caller
                                     of establisher }
     END;
   Handler_2 := SS$_RESIGNAL;
   END;
```

A handler can perform a default unwind to force return to the caller of its establisher. If the establisher is a function whose result is expected in R0 or R0 and R1, the handler must establish the return value by modifying the third or third and fourth positions of the mechanism array (the locations of the return R0 and R1 values). If the establisher is a function whose result is returned by the extra-parameter method, the handler must establish the condition value by assignment to the function identifier. In this case, you must observe two additional restrictions:

- The handler must be nested within the function
- The function result must be declared VOLATILE

Example 4

```
[ASYNCHRONOUS] FUNCTION Handler_3
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;
```

```
BEGIN
IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
THEN
   BEGIN
       .
       .                        { cleanup }
       .
   MA[3] := expression;         { establish function result
                                  seen by caller }
   $UNWIND (MA[2]);             { unwind to establisher }
   END;
Handler_3 := SS$_RESIGNAL;
END;
```

A handler can also force return to its establisher immediately following the point of call. In this case, you should make sure that the handler understands whether the currently uncompleted call was a function call (in which case a returned value is expected) or a procedure call. If the uncompleted call is a function call that will return a value in R0 or R0 and R1, then the handler can modify the mechanism array to supply a value. If, however, the uncompleted call is a function call that will return a value using the extra-parameter mechanism, then there is no way for the handler to supply a value.

Example 5

```
[ASYNCHRONOUS] FUNCTION Handler_4
   (VAR SA : Sig_Args;
    VAR MA : Mech_Args)    : [UNSAFE] INTEGER;

   BEGIN
   IF LIB$MATCH_COND (SA[1],   condition-name   ,...) <> 0
   THEN
      GOTO 99;
   Handler_4 := SS$_RESIGNAL;
   END;
```

A handler can force control to resume at an arbitrary label in its scope. Note that this reference is to a label in an enclosing block, because a GOTO to a local label would simply remain within the handler. In accordance with the VAX Procedure Calling Standard, VAX PASCAL implements references to labels in enclosing blocks by signaling SS$_UNWIND in all stack frames that must be deleted.

Example 6

```
FUNCTION EXP_With_Status
   (X : REAL;
    VAR Status : Cond_Status)     : REAL;

   FUNCTION MTH$EXP
      (A : REAL)
         : REAL;
      EXTERNAL;

   [ASYNCHRONOUS] FUNCTION Math_Error
      (VAR SA : Sig_Args;
       VAR MA : Mech_Args)       : [UNSAFE] INTEGER;

      BEGIN   { Math_Error }
      IF LIB$MATCH_COND (SA[1], MTH$_FLOOVEMAT, MTH$_FLOUNDMAT) <> 0
      THEN
         BEGIN
         IF Status.STS$_SUCCESS   { record condition status
                                    if no previous error }
         THEN
             Status := SA[1]::Cond_Status;
             Math_Error := SS$_CONTINUE;  { condition handled,
                                             propagate no further }
         END
       ELSE
          Math_Error := SS$_RESIGNAL;  { propagate condition status
                                         to other handlers }
          END;

   BEGIN   { EXP_With_Status }
   STATUS := SS$_SUCCESS;
   ESTABLISH (Math_Error);
   EXP_With_Status := MTH$EXP (X);
   END;
```

This example shows a handler that records the condition status if a floating overflow or underflow error is detected during the execution of the mathematical function MTH$EXP.

Example 7

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Use_A_Handler(INPUT,OUTPUT);

TYPE
   Sigarr  = ARRAY [0..9] OF INTEGER;
   Mecharr = ARRAY [0..4] OF INTEGER;

VAR
   F1,F2 : REAL;
```

```
[ASYNCHRONOUS] FUNCTION My_Handler(
      VAR Sigargs  : Sigarr;
      VAR Mechargs : Mecharr) : INTEGER;

  VAR
    Outfile : TEXT;

  [ASYNCHRONOUS] FUNCTION LIB$FIXUP_FLT(
    VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr;
        New_Opnd : REAL := %IMMED 0) : INTEGER; EXTERNAL;

  [ASYNCHRONOUS] FUNCTION LIB$SIM_TRAP(
    VAR Sigargs  : Sigarr;
    VAR Mechargs : Mecharr) : INTEGER; EXTERNAL;

    BEGIN
    OPEN(Outfile,'TT:');
    REWRITE(Outfile);

    { Handle various conditions }
    CASE Sigargs[1] OF

    { Convert floating faults to traps }
    SS$_FLTDIV_F, SS$_FLTOVF_F :
       LIB$SIM_TRAP(Sigargs,Mechargs);

    { Handle the floating divide by zero trap }
    SS$_FLTDIV :
       BEGIN
       WRITELN(Outfile,'Floating divide by zero');
       My_Handler := SS$_CONTINUE;
       END;

    { Handle the floating overflow trap }
    SS$_FLTOVF :
       BEGIN
       WRITELN(Outfile,'Floating overflow');
       My_Handler := SS$_CONTINUE;
       END;

    { Handle taking the square root }
    MTH$_SQUROONEG :
        BEGIN
        WRITELN(Outfile,'Square root of a negative number');
        My_Handler := SS$_CONTINUE;
        END;

    { Handle the reserved operand left by SQRT }
    SS$_ROPRAND :
        BEGIN
        WRITELN(Outfile,'Reserved floating operand');
        LIB$FIXUP_FLT(Sigargs,Mechargs);
        My_Handler := SS$_CONTINUE;
        END;
      OTHERWISE
         BEGIN
         WRITELN(Outfile,'Condition occurred, ',HEX(Sigargs[1]));
         My_Handler := SS$_RESIGNAL;
         END;

      END;

      CLOSE(Outfile);

      END;

BEGIN
ESTABLISH(My_Handler);
F1 := 0.0;
F2 := 1E38;

{ Generate exception conditions }
F1 := F2 / 0.0;
F1 := F2 * f2;
F1 := SQRT(-1.0);
END.
```

This example demonstrates the use of VAX/VMS condition handlers with VAX PASCAL.

Chapter 8

Calling Conventions

This chapter describes how to call routines that are not written in VAX PASCAL and provides information on calling VAX/VMS system routines. See the *VAX PASCAL Reference Manual* for information on declaring and calling VAX PASCAL routines.

The *Introduction to VAX/VMS System Routines* contains detailed information about procedure-calling and argument-passing mechanisms. You should be familiar with these subjects before you attempt to use the features described here. Note that the *Introduction to VAX/VMS System Routines* uses the term "procedure" to mean any routine entered by a CALL instruction. This chapter uses the term "routine" instead, to avoid confusion with VAX PASCAL's definition of "procedure."

8.1 VAX Procedure Calling Standard

Programs compiled by the VAX PASCAL compiler conform to the standard defined for VAX procedure calls (see the *Introduction to VAX/VMS System Routines*). This standard describes how parameters are passed, how function values are returned, and how routines receive and return control. By means of the calling standard, VAX PASCAL provides features that allow programs to call system routines written in other native-mode languages supported by the VAX/VMS system.

8.1.1 Parameter Lists

Each time a routine is called, the VAX PASCAL compiler constructs a parameter list. The VAX Procedure Calling Standard defines a parameter list as a sequence of longword (4-byte) entries. The first byte of the first entry in the list is a parameter count, which indicates how many parameters follow in the list.

The form in which the parameters in the list are represented is determined by the passing mechanisms you specify in the formal parameter list and the values you pass in the actual parameter list. The parameter list contains the actual parameters passed to the routine.

8.1.2 Function Return Values

In VAX PASCAL, a function returns to the calling block the value that was assigned to its identifier during execution. The compiler chooses one of three methods for returning this value; the method chosen depends on the amount of storage required for values of the type returned.

- If the value can be represented in 32 bits of storage, it is returned in register R0.
- If the value requires from 33 to 64 bits, the low-order bits of the result are returned in register R0 and the high-order bits are returned in register R1.
- If the value is too large to be represented in 64 bits or if its type is a string type (PACKED ARRAY OF CHAR or VARYING OF CHAR), the calling routine allocates the required storage. An extra parameter—a pointer to the location where the function result will be stored—is added to the beginning of the calling routine's actual parameter list.

For values of structured types, the amount of storage required for the entire structure determines which of the three return methods is used. Character strings are always returned by the extra-parameter method.

Note that functions that require the use of an extra parameter can have no more than 254 parameters; functions that store their results in registers R0 and R1 can have 255 parameters.

Table 8–1 lists the methods by which values of each type are returned.

Table 8–1: Function Return Methods

Type	Return Method
INTEGER, UNSIGNED, CHAR, BOOLEAN, REAL, SINGLE, pointer, enumerated, subrange	General register R0
DOUBLE	R0: Low-order result R1: High-order result
QUADRUPLE, VARYING OF CHAR, PACKED ARRAY OF CHAR	Extra-parameter
Other structured types	Depends on the amount of storage required

8.1.3 Contents of the Call Stack

A call stack is a temporary area of storage allocated by the system for each user process. On the call stack, the system maintains information about each routine call in the current image. Each time a routine is called by a VAX PASCAL program, the hardware creates a structure on the call stack; this structure is known as the call frame. The call frame for each active routine contains the following:

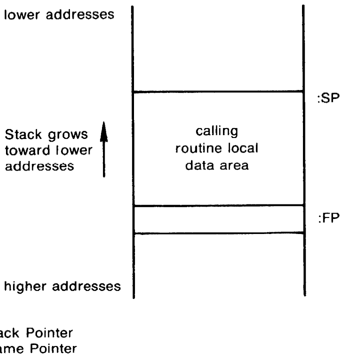
- A pointer to the call frame of the previous routine call. This pointer is called the saved frame pointer (FP).
- The saved argument pointer (AP) of the previous routine call.
- The storage address of the point at which the routine was called; that is, the address of the instruction following the call to the current routine. This address is called the saved program counter (PC).
- The saved contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

When execution of a routine ceases, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

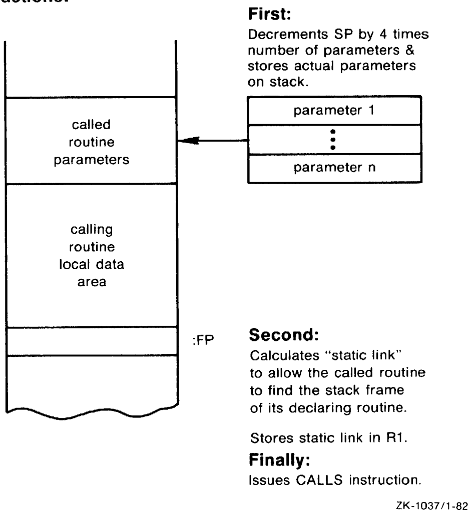
The VAX PASCAL compiler uses the VAX CALLS and CALLG instructions to call routines. Figure 8–1 illustrates the events that occur during a routine call and shows the structure of the call stack after each event.

Figure 8–1: Contents of the Run-Time Stack

1 Before routine call:



2 The calling routine's actions:

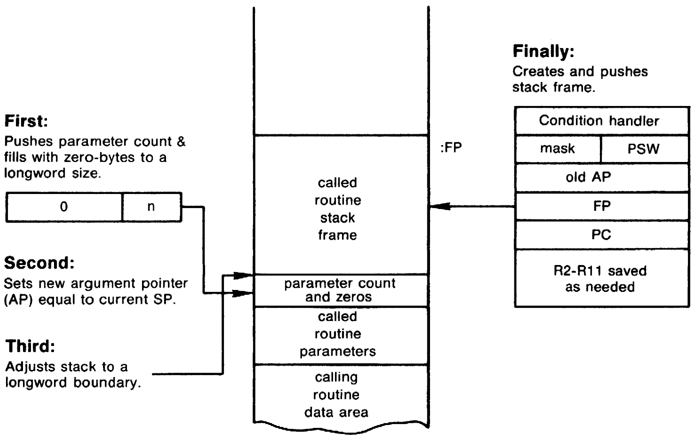


First:
Decrements SP by 4 times number of parameters & stores actual parameters on stack.
parameter 1
parameter n
called routine parameters
calling routine local data area
:FP
Second:
Calculates "static link" to allow the called routine to find the stack frame of its declaring routine.
Stores static link in R1.
Finally:
Issues CALLS instruction.
ZK-1037/1-82

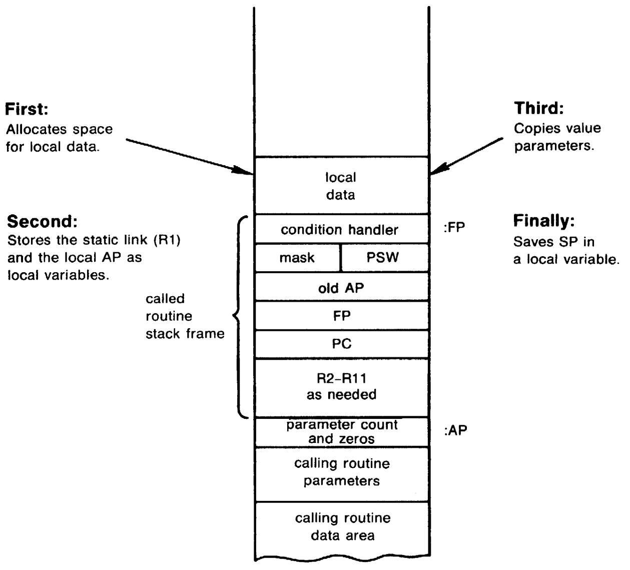
(Continued on next page)

Figure 8–1 (Cont.): Contents of the Run-Time Stack

❸ **The CALLS instruction's actions:**


First:
Pushes parameter count & fills with zero-bytes to a longword size.
0
n
Second:
Sets new argument pointer (AP) equal to current SP.
Third:
Adjusts stack to a longword boundary.
called routine stack frame
:FP
parameter count and zeros
called routine parameters
calling routine data area
Finally:
Creates and pushes stack frame.
Condition handler
mask
PSW
old AP
FP
PC
R2-R11 saved as needed

❹ **The called routine's actions:**


First:
Allocates space for local data.
Second:
Stores the static link (R1) and the local AP as local variables.
called routine stack frame
local data
condition handler
mask
PSW
old AP
FP
PC
R2-R11 as needed
parameter count and zeros
calling routine parameters
calling routine data area
Third:
Copies value parameters.
:FP
Finally:
Saves SP in a local variable.
:AP

ZK-1037/2-82

As shown in Figure 8–1, the frame pointer of the calling routine is stored in R1, thus creating a link between the calling routine and the called routine. Because this link exists, the called routine can access variables and routines declared in enclosing blocks and can use GOTO statements to access executable statements in enclosing blocks.

However, if you declare a routine with the UNBOUND attribute, the system does not assume that the frame pointer of the declaring routine is stored in R1; thus, no link is established. As a result, an unbound routine has the following restrictions:

- It cannot access automatic variables declared in enclosing blocks.
- It cannot call bound routines declared in enclosing blocks.
- It cannot use a GOTO statement to transfer control to enclosing blocks other than the main program block.

By default, routines declared at program or module level and all other routines declared with the INITIALIZE, GLOBAL, or EXTERNAL attributes have the characteristics of unbound routines. Routines passed by the immediate value mechanism (see Section 8.3.1) must be UNBOUND.

Asynchronous system trap routines (ASTs) and RMS completion routines must have both the ASYNCHRONOUS and UNBOUND attributes. Because they are asynchronous, such routines can access only volatile variables, predeclared routines, and other asynchronous routines. Note that the VAX PASCAL run-time system does not permit a program and an asynchronous routine (such as an AST) to access the same file simultaneously. See the *VAX PASCAL Reference Manual* for more information on the ASYNCHRONOUS and UNBOUND attributes.

8.2 Parameter-Passing Semantics

The parameter-passing semantics describe how parameter values are passed between the calling and called routine. The Pascal standard defines two parameter-passing semantics: value passing semantics and variable passing semantics. In addition, VAX PASCAL defines another type of parameter-passing semantics, foreign passing semantics, used to pass parameters to routines that are not written in VAX PASCAL. By default, VAX PASCAL passes arguments using value semantics. Table 8–2 lists the VAX PASCAL mechanisms that can be used to obtain the various semantics.

Table 8–2: Parameter-Passing Semantics

Parameter-Passing Semantics	Methods Used in VAX PASCAL
Value	Default
Variable	VAR, %REF, %DESCR, %STDESCR, [REFERENCE], [CLASS_A], [CLASS_NCA], or [CLASS_S]
Foreign	%IMMED, %REF, %DESCR, %STDESCR, [REFERENCE], [IMMEDIATE], [CLASS_A], [CLASS_NCA], or [CLASS_S]

You can obtain different parameter-passing semantics when using the same VAX PASCAL method. For example, the %REF passing mechanism can result in variable passing semantics or foreign passing semantics depending, on the actual parameter and the formal parameter. This topic is covered in greater detail in the following sections.

8.2.1 Value Passing Semantics

When you specify value passing semantics, the address of the actual parameter is passed to the called routine. The value from the specified address is then copied from the specified address to the called routine's local storage. This method should be used only when the called routine does not need to change the value of the actual parameter.

When you use value semantics, the actual parameter can be a compile-time or run-time expression; the names of routines are not allowed as value parameters.

For example, the following function declaration requires an actual parameter to be passed by reference with value passing semantics:

```
[EXTERNAL] Function MTH$TANH( Angle      : REAL ) : REAL; EXTERN;
     .
     .
     .
Hyperbolic_Tan := MTH$TANH( Radians );
```

The function MTH$TANH returns, as a real value, the hyperbolic tangent of its parameter. The input parameter to this function is the size of the angle in radians. The value, which is not changed by MTH$TANH, is passed with value passing semantics.

8.2.2 Variable Passing Semantics

When you specify variable passing semantics, the address of the actual parameter is passed to the called routine. In contrast to value passing semantics, however, the called routine directly accesses the actual parameter. Thus, execution of the called routine can change the value of the actual parameter.

When you use variable passing semantics, the actual parameter must be a variable or a component of an unpacked structured variable; no other expressions are allowed unless the formal parameter has the READONLY attribute. The names of routines are not allowed as variable parameters.

You must use variable passing semantics when passing a file variable as an actual parameter. This method is also useful for preventing the copying of large parameters if they were passed by value semantics.

The following function declaration requires an actual parameter to be passed by reference with variable passing semantics:

```
TYPE
    $Quad = [QUAD,UNSAFE] RECORD
    LO : UNSIGNED;
    L1 : INTEGER;
    END;

VAR
    Systime : $Quad;

[ASYNCHRONOUS,EXTERNAL(SYS$GETTIM)] FUNCTION $GETTIM
    ( VAR TIMADDR : [VOLATILE] $QUAD ) : INTEGER; EXTERN;
    .
    .
    .
Status := $GETTIM( Systime );
```

This example declares an external routine, SYS$GETTIM, which returns the system time. The input parameter is a 64-bit variable into which the system service writes the time. Because the function changes the value of the actual parameter Systime, Systime is passed by variable semantics.

8.2.3 Foreign Passing Semantics

VAX PASCAL also defines another passing semantics, known as foreign passing semantics. This method allows for more flexibility when a program calls external routines, that is, routines not written in VAX PASCAL.

Under the rules of true foreign passing semantics, the calling routine makes a copy of the actual parameter's value and passes the address of the copy to the called routine. Although the called routine is allowed to change this value, a change is not reflected when control returns to the calling routine. Note how foreign passing semantics differ from value passing semantics: with foreign passing semantics, the copy is made by the calling (VAX PASCAL) routine, whereas with value passing semantics, the copy is made by the called (external) routine.

8.3 Parameter-Passing Mechanisms

The way in which an argument specifies the actual data to be used by the called routine is defined by the parameter-passing mechanism. In compliance with the VAX Procedure Calling Standard, VAX PASCAL supports the three basic passing mechanisms:

- By immediate value—the longword argument in the argument list contains the actual data.
- By reference—the longword argument in the argument list contains the address of the data to be used by the routine.
- By descriptor—the longword argument in the argument list contains the address of a descriptor that describes the location, length, and data type of the data to be used by the routine.

By default, VAX PASCAL uses the by reference mechanism to pass all actual parameters except those that correspond to conformant parameters, in which case the by descriptor mechanism is used. Table 8–3 describes the method you use in VAX PASCAL to obtain the desired parameter-passing mechanism.

Table 8–3: Parameter-Passing Mechanisms

Parameter-Passing Mechanism	Methods Used in VAX PASCAL
By immediate value	%IMMED or [IMMEDIATE]
By reference	Default or %REF
By descriptor	%DESCR, %STDESCR, [CLASS_S], [CLASS_A], or [CLASS_NCA]

A mechanism specifier usually appears before the name of a formal parameter, or if a passing attribute is used it appears in the attribute list of the formal parameter. However, in VAX PASCAL, a mechanism specifier can also appear before the name of an actual parameter. In the latter case, the specifier overrides the type, passing semantics, and passing mechanism specified in the formal parameter declaration. This topic is discussed more thoroughly in Section 8.3.5.

8.3.1 By Immediate Value Passing Mechanism

The by immediate value passing mechanism passes a copy of a value instead of the address. VAX PASCAL provides the %IMMED foreign passing mechanism and the IMMEDIATE attribute in order to pass a parameter by immediate value.

8.3.1.1 %IMMED Mechanism Specifier and IMMEDIATE Attribute

The by immediate value mechanism passes the actual data as the argument. By using either the %IMMED mechanism specifier or the IMMEDIATE attribute, you can obtain the by immediate value passing mechanism. Use of the %IMMED mechanism specifier or the IMMEDIATE attribute always implies immediate foreign passing semantics.

When you use the %IMMED mechanism specifier or the IMMEDIATE attribute, the compiler passes a value rather than an address. Only formal value and routine parameters declared in external routines can have the %IMMED specifier or IMMEDIATE attribute. The actual parameter can be a compile-time or run-time expression, or a routine identifier.

In addition, because a by immediate value argument is only one longword in length, variables that require more than 32 bits of storage, including file variables, cannot be passed by immediate value.

The following function declaration requires an actual parameter to be passed by immediate value:

```
[ASYNCHRONOUS, EXTERNAL(SYS$WAITFR)] FUNCTION $WAITFR
    ( %IMMED EFN : INTEGER ) : INTEGER; EXTERNAL;

VAR
    Event_Flag : INTEGER;
    .
    .
    .
Status := $WAITFR( Event_Flag );
```

The external routine SYS$WAITFR, which waits for a specific event flag, requires one input 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.

Normally, an actual procedure or function parameter is passed as the address of a bound procedure value. The VAX architecture's bound procedure value's data type consists of two longword entries. The first entry is the address of the entry mask. The second entry is the routine's static scope pointer, the address of the call frame in which the routine was declared. However, when a routine is passed by immediate value, the argument list contains only one entry, the address of the entry mask. No static scope pointer is passed; therefore, any routine passed to an immediate parameter should access only its local variables and variables with static allocation. For more information on bound procedure values, see the *Introduction to VAX/VMS System Routines.*

The following procedure declaration requires a procedure parameter to be passed by immediate value:

```
[EXTERNAL] PROCEDURE Forcaller( [IMMEDIATE,UNBOUND] PROCEDURE Utility );
FORTRAN;

PROCEDURE Read_Char;
    BEGIN
    .
    .
    .
    END;
    .
    .
    .
Forcaller( Read_Char );
```

The VAX PASCAL procedure Read_Char is passed by immediate value to the VAX FORTRAN subroutine Forcaller. The argument list contains the address of Read_Char's entry mask, which indicates the registers to be saved.

8.3.2 By Reference Passing Mechanism

The by reference mechanism passes the address of the parameter to the called routine. This is the default parameter passing mechanism. When using the default by reference passing mechanism, the type of passing semantics used depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

In addition to using the defaults, the VAX PASCAL compiler provides the %REF foreign passing mechanism and the REFERENCE attribute, which has more than one interpretation for the passing semantics depending on the data item represented by the actual parameter. This allows you the flexibility to have the called routine use either variable semantics or true foreign semantics. The mechanism specifier usually appears before the name of a formal parameter. The parameter passing attribute appears in the attribute list of the formal parameter.

8.3.2.1 %REF Mechanism Specifier and REFERENCE Attribute

When you use the %REF mechanism specifier or the REFERENCE attribute, the parameter is passed by reference. That is, the address of the parameter is passed to the called routine. The %REF mechanism specifier or the REFERENCE attribute can be used on all VAX PASCAL data types. The passing semantics used depends on the data item represented by the actual parameter.

- If the actual parameter is a variable, %REF or [REFERENCE] on the formal parameter implies variable semantics within the called routine.
- If the actual parameter is an expression or a variable enclosed in parentheses, %REF or [REFERENCE] implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following procedure declaration requires two parameters passed by reference:

```
TYPE
   Matrix = ARRAY [1..10] OF REAL;
VAR
   A,B,Q  : Matrix;

[EXTERNAL]PROCEDURE Find_Determinant( Det  : [REFERENCE] Matrix;
                              VAR Inv  : Matrix ); EXTERN;
```

The procedure Find_Determinant performs a matrix inversion and calculates the determinant. The following examples are legal calls to Find_Determinant:

```
Find_Determinant( A, Q );
Find_Determinant( (A), Q );
```

In the first call, A is passed by reference using variable semantics. In the second call, foreign semantics are used and the calling routine copies the value of A and passes the address of the copy to Find_Determinant. In both calls, Q is passed by reference using variable semantics.

8.3.3 By Descriptor Passing Mechanism

There are several types of descriptors. Each descriptor contains a value that identifies the descriptor's type. The called routine can then access the information held in the descriptor to identify its type.

When you use one of the VAX PASCAL by descriptor mechanisms, the compiler passes the address of a string, array, or scalar descriptor, as described in the *Introduction to VAX/VMS System Routines*. The VAX PASCAL compiler generates the descriptor supplying the necessary information.

VAX PASCAL provides three attributes for the by descriptor passing mechanism: [CLASS_S], [CLASS_A] and [CLASS_NCA]. With these three attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used. The parameter-passing attribute appears in the attribute list of the formal parameters.

Sometimes you may want the flexibility of choosing the passing semantics, either variable semantics or true foreign semantics. In this instance, the VAX PASCAL compiler provides two foreign passing mechanism specifiers, %DESCR and %STDESCR. These specifiers have more than one interpretation for the passing semantics depending on the data type of the actual parameter. The mechanism specifier usually appears before the name of a formal parameter.

Table 8–4 lists the class and type of descriptor generated for parameters that can be passed using the by descriptor mechanism. See the *Introduction to VAX/VMS System Routines* for complete information on descriptor classes and types.

Table 8–4: Parameter Descriptors

Parameter Type	Descriptor Class and Type		
	%DESCR	%STDESCR	Value or VAR Semantics
Ordinal	DSC$K_CLASS_S[1]	—	—
SINGLE	DSC$K_CLASS_S DSC$K_DTYPE_F	—	—
DOUBLE	DSC$K_CLASS_S DSC$K_DTYPE_D or DSC$K_DTYPE_G	—	—
QUADRUPLE	DSC$K_CLASS_S DSC$K_DTYPE_H	—	—
RECORD	—	—	—
ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	—
ARRAY OF VARYING OF CHAR	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	—
Conformant ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]	DSC$K_CLASS_A[2]
Conformant ARRAY OF VARYING OF CHAR[4]	DSC$K_CLASS_VSA DSC$K_DTYPE_VT	—	DSC$K_CLASS_VSA DSC$K_DTYPE_VT
VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	—

[1]Descriptor's D_type depends on size of type

[2]Descriptor's D_Type depends on component type

[3]Only if PACKED ARRAY OF CHAR

[4]Component type can be a conformant VARYING schema

Table 8–4 (Cont.): Parameter Descriptors

Parameter Type	Descriptor Class and Type		
	%DESCR	**%STDESCR**	**Value or VAR Semantics**
Conformant VARYING OF CHAR	DSC$K_CLASS_VS DSC$K_DTYPE_VT	—	DSC$K_CLASS_VS DSC$K_DTYPE_ VT
SET	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	—
FILE	DSC$K_CLASS_S DSC$K_DTYPE_Z	—	
Pointer	DSC$K_CLASS_S DSC$K_DTYPE_LU	—	—
PROCEDURE or FUNCTION	DSC$K_CLASS_S DSC$K_DTYPE_BPV	—	Bound procedure value by reference
	CLASS_A	**CLASS_NCA**	**CLASS_S**
Ordinal	—	—	DSC$K_CLASS_S[1]
SINGLE	—	—	DSC$K_CLASS_S DSC$K_CLASS_S DSC$K_DTYPE_F
DOUBLE	—	—	DSC$CLASS_S DSC$DTYPE_D or DSC$DTYPE_G
QUADRUPLE	—	—	DSC$CLASS_S DSC$DTYPE_H
RECORD	—	—	—
ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC$K_DTYPE_ T[3]

[1]Descriptor's D_type depends on size of type

[2]Descriptor's D_Type depends on component type

[3]Only if PACKED ARRAY OF CHAR

Table 8–4 (Cont.): Parameter Descriptors

Parameter Type	Descriptor Class and Type		
	CLASS_A	CLASS_NCA	CLASS_S
ARRAY OF VARYING OF CHAR	—	—	—
Conformant ARRAY	DSC$K_CLASS_A[2]	DSC$K_CLASS_NCA[2]	DSC$K_CLASS_S DSC$K_DTYPE_T[3]
Conformant ARRAY OF VARYING OF CHAR[4]	—	—	—
VARYING OF CHAR	—	—	—
Conformant VARYING OF CHAR	—	—	—
SET	—	—	DSC$K_CLASS_S DSC$K_DTYPE_Z
FILE	—	—	DSC$K_CLASS_S DSC$K_DTYPE_Z
Pointer	—	—	DSC$K_CLASS_S DSC$K_DTYPE_LU
PROCEDURE or FUNCTION	—	—	—

[2]Descriptor's D_Type depends on component type

[3]Only if PACKED ARRAY OF CHAR

[4]Component type can be a conformant VARYING schema

8.3.3.1 CLASS_S Attribute

When the CLASS_S attribute is used on a formal parameter, the compiler generates a fixed-length scalar descriptor of a variable and passes its address to the called routine. Only ordinal, real, set, pointer, and one-dimensional arrays (packed or unpacked, fixed or conformant) that are of type OF CHAR can have the CLASS_S attribute on the formal parameter.

With the CLASS_S attribute, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

8.3.3.2 CLASS_A and CLASS_NCA Attributes

When you use the CLASS_A or CLASS_NCA attribute on a formal parameter, the compiler generates an array descriptor and passes its address to the called routine. The type of the CLASS_A and CLASS_NCA parameter must be an array (packed or unpacked, fixed or conformant) of an ordinal or real type.

With the CLASS_A and CLASS_NCA attributes, the type of passing semantics used for the by descriptor argument depends on the use of the VAR keyword. If the formal parameter name is preceded by the reserved word VAR, variable semantics is used; otherwise, value semantics is used.

8.3.3.3 %STDESCR Mechanism Specifier

When you use the %STDESCR mechanism specifier, the compiler generates a fixed-length descriptor of a character-string variable and passes its address to the called routine. Only items of the following types can have the %STDESCR specifier on the actual parameter: character-string constants, string expressions, packed character arrays with lower bounds of 1, and packed conformant arrays with indexes of an integer or integer subrange type. The passing semantics used depends on the variable represented by the actual parameter:

- If the actual parameter is a variable of type PACKED ARRAY OF CHAR, %STDESCR implies variable semantics within the called routine.
- If the actual parameter is either a variable enclosed in parentheses, an expression, or a VARYING OF CHAR variable, %STDESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires one fixed-length string descriptor as a parameter:

```
[ASYNCHRONOUS,EXTERNAL(SYS$SETDDIR)] FUNCTION $SETDDIR
        (%STDESCR New_dir : PACKED ARRAY [$L1..$U1: INTEGER] OF CHAR;
         Old_Dir_Len : $UWORD := %IMMED 0;
         Old_Dir : VARYING [LM] OF CHAR : %IMMED 0)
                : INTEGER; EXTERN;

 .
 .
 .
Status :=  $SETDDIR('[VAX_PASCAL]');
```

The actual parameter '[VAX_PASCAL]' is passed by string descriptor with foreign semantics to the formal parameter New_Dir.

8.3.3.4 %DESCR Mechanism Specifier

When you use the %DESCR mechanism specifier, the parameter generates a descriptor for an ordinal, real or array variable and passes its address to the called routine. The type of %DESCR parameter can be any ordinal or real type, a VARYING OF CHAR string, or an array (packed or unpacked, fixed or conformant) of an ordinal or real type. External routines that are written in high-level languages or contained in the VAX/VMS Run-Time Library often require such descriptors.

The passing semantics used depends on the actual parameter's data type:

- If the actual parameter is a variable, %DESCR formal parameter implies variable semantics within the called routine.
- If the actual parameter is an expression or a variable enclosed in parentheses, %DESCR implies foreign semantics.

If the actual parameter is not modified by the called external routine, the corresponding formal parameter should be declared READONLY, saving the copy from having to be made.

The following function declaration requires a varying-length string descriptor as its parameter.

```
TYPE
    Vary = VARYING [30] OF CHAR;
VAR
    Obj_String  : Vary;
    Times_Found : INTEGER;

[EXTERNAL] FUNCTION Search_String( %DESCR String_Val : Vary ) : BOOLEAN;
EXTERNAL;

 .
 .
 .
IF Search_String( Obj_String )
THEN
Times_Found := Times_Found + 1;
```

The actual parameter Obj_String is passed by varying string descriptor with variable semantics to the formal parameter String_Val.

8.3.4 Mechanism Specifiers on Actual Parameters

When calling an external routine, you must pass actual parameters by the mechanism stated or implied in the routine declaration. VAX PASCAL allows you to use the foreign mechanism specifiers %IMMED, %REF, %DESCR and %STDESCR before an actual parameter in a routine call.

When a specifier appears on an actual parameter, it overrides the type, semantics, and any mechanism specified on the formal parameter declaration. Thus, type checking is suspended for the parameter association to which the specifier applies.

Regardless of whether the mechanism is determined by a formal or an actual parameter, the mechanism specifier is interpreted in the same way.

Special considerations arise when a function that has no formal parameters of its own (or that has defaults that are being used for all of its formal parameters) is passed as an actual parameter to another routine. The appearance of the function identifier in an actual parameter list could indicate the passing of either the address of the entry mask or the function result. For example:

```
[EXTERNAL] PROCEDURE Proc1
   (J  : REAL); EXTERNAL;

FUNCTION I
   : INTEGER;
     .
     .
     .
Proc1 (%REF I):
```

Because the compiler does not perform type checking when function I is passed to procedure Proc1, it is unclear whether this call passes the function or executes it and then passes the result. (Note that the appearance of %REF before the actual parameter I inhibits the compiler from converting I to type REAL.)

The compiler resolves the ambiguity as follows:

- When the function identifier is enclosed in parentheses, the function is executed and the result is passed by the specified mechanism.
- When the function identifier is not enclosed in parentheses, the address of the function's entry mask is passed by the specified mechanism.

Thus, function I in the above example is passed by reference. The following call, in which the function identifier is enclosed in parentheses, passes the result of I by reference:

```
Proc1 (%REF (I));
```

A similar ambiguity arises when a mechanism specifier precedes the name of a variable in an actual parameter list. The %IMMED mechanism specifier always causes a value to be passed. However, %REF, %STDESCR, and %DESCR either cause the value to be accessed directly or cause a local copy to be made. For example:

```
TYPE
   Arr_type = PACKED ARRAY[1..10] OF CHAR;

VAR
   Arr : Arr_Type

[EXTERNAL] PROCEDURE Proc2
   (Arr2 : Arr_Type); EXTERNAL;
   .
   .
   .
Proc2 (%DESCR Arr);
```

Again, the suspension of type checking makes it unclear whether this call passes the actual variable or a local copy.

The ambiguity is resolved as follows:

- When the variable identifier is enclosed in parentheses, a copy of the variable is passed by the specified mechanism.
- When the variable identifier is not enclosed in parentheses, the variable itself is accessed by the specified mechanism.

Thus, the array Arr in the example above is passed by descriptor. The following call, in which the variable identifier is enclosed in parentheses, passes a copy of Arr by the same mechanism.

```
Proc2 (%DESCR (Arr));
```

8.3.5 Summary of Passing Mechanisms and Passing Semantics

Table 8–5 summarizes the passing semantics used when various mechanisms are specified on either the formal or the actual parameter. For example, if a variable is passed to a formal parameter that was declared without the keyword VAR and either %REF or [REFERENCE] was specified, then variable passing semantics will be used. Likewise, if a variable is passed to a formal parameter which was declared with the keyword VAR and either %REF or [REFERENCE] was specified, then an error will occur.

Note that if an actual parameter is preceded by a %IMMED specifier, regardless of what passing mechanism is used to declare the formal parameter, foreign semantics would be used, because a specifier appearing on the actual parameter always overrides the semantics specified on the formal parameter.

Table 8–5: Summary of Passing Mechanisms and Passing Semantics

	Actual Parameter			
	Variable		(Variable) or Expression	
Passing Mechanism	No VAR on Formal	VAR on Formal	No VAR on Formal	VAR on Formal
By immediate value				
%IMMED or [IMMEDIATE]	Foreign	Error	Foreign	Error
By reference				
default	Value	Variable	Value	Value[1]
%REF or [REFERENCE]	Variable	Error	Foreign	Error

[1]If the formal parameter is declared with the READONLY attribute, then value passing semantics is used; otherwise, it is an error.

Table 8–5 (Cont.): Summary of Passing Mechanisms and Passing Semantics

	Actual Parameter			
	Variable		(Variable) or Expression	
Passing Mechanism	No VAR on Formal	VAR on Formal	No VAR on Formal	VAR on Formal
By descriptor				
[CLASS_S]	Value	Variable	Value	Value[1]
[CLASS_A]	Value	Variable	Value	Value[1]
[CLASS_NCA]	Value	Variable	Value	Value[1]
%STDESCR	Variable	Error	Foreign	Error
%DESCR	Variable	Error	Foreign	Error

[1]If the formal parameter is declared with the READONLY attribute, then value passing semantics is used; otherwise, it is an error.

8.4 Passing Parameters from VAX PASCAL to Non-VAX PASCAL Routines

Non-VAX PASCAL routines require parameters in the form of addresses, immediate values, or descriptors. As described in the previous sections, VAX PASCAL provides various ways to pass parameters to the called routine. You can specify the desired passing mechanism by using one of the mechanism specifiers %REF, %IMMED, %DESCR, or %STDESCR, or by specifying one of the passing mechanism attributes [REFERENCE], [IMMEDIATE], [CLASS_S], [CLASS_A], or [CLASS_NCA].

You can also specify the by reference mechanism by indicating value and variable semantics in the declarations of formal parameters to non-PASCAL routines. Table 8–6 illustrates the types of parameters that can be passed to foreign mechanism parameters.

Table 8–6: Foreign Mechanism Parameters

	Passing Mechanism						
Parameter Type	%IMMED	%REF	%DESCR	%STDESCR	CLASS_S	CLASS_A	CLASS_NCA
Ordinal	Yes	Yes	Yes	No	Yes	No	No
SINGLE	Yes	Yes	Yes	No	Yes	No	No
DOUBLE or QUADRUPLE	No	Yes	Yes	No	Yes	No	No
RECORD	Yes[1]	Yes	No	No	No	No	No
ARRAY	Yes[1]	Yes	Yes	Yes[2]	Yes[2]	Yes	Yes
ARRAY OF VARYING OF CHAR	No	Yes	Yes	No	No	No	No
Conformant ARRAY	Yes[1]	Yes	Yes	Yes[2]	Yes[2]	Yes	Yes
Conformant ARRAY OF VARYING OF CHAR[3]	No	Yes	Yes	No	No	No	No
VARYING OF CHAR	No	Yes	Yes	No	No	No	No
Conformant VARYING OF CHAR	No	Yes	Yes	No	No	No	No
SET	Yes[1]	Yes	Yes	No	Yes	No	No
FILE	No	Yes	No	No	No	No	No
Pointer	Yes	Yes	Yes	No	Yes	No	No
PROCEDURE or FUNCTION	Yes[4]	Yes	Yes	No	No	No	No

[1]Allocation size must be less than or equal to 32 bits.

[2]Only if PACKED ARRAY OF CHAR.

[3]Component type can be a conformant VARYING schema.

[4]Must be unbound.

8.5 Passing Parameters from Non-VAX PASCAL to VAX PASCAL Routines

When calling a VAX PASCAL routine from a non-VAX PASCAL routine, you must ensure that the parameters are in the form required by the VAX PASCAL routine. By default, VAX PASCAL requires most parameters to be passed by reference.

- When the VAX PASCAL routine requires a value parameter, the parameter list of the calling routine must contain the address of a value. The VAX PASCAL routine will copy the value from the passed address upon entry.
- When the VAX PASCAL routine requires a VAR parameter, the parameter list of the calling routine must contain the address of a variable. The VAX PASCAL routine uses the address to access the actual parameter variable. An actual parameter variable whose value can change as a result of routine execution must be passed in this manner. In addition, all files must be passed to VAX PASCAL routines as VAR parameters.
- When the VAX PASCAL routine requires a formal procedure or function parameter, the parameter list of the calling routine must specify the address of a bound procedure value. This process implements the VAX by reference mechanism for a routine.
- When the VAX PASCAL routine requires a formal conformant array, the parameter list of the calling routine must contain the address of a CLASS_A descriptor.
- When the VAX PASCAL routine requires a formal conformant VARYING parameter, the parameter list of the calling routine must contain the address of a CLASS_VS descriptor.

8.6 VAX/VMS System Routines

To eliminate duplication of programming and debugging efforts, the VAX/VMS system provides many routines to perform common programming tasks. These routines are collectively known as system routines. They include routines in the VAX/VMS Run-Time Library to assist you in such areas as mathematics, screen management, and string manipulation. Also included are VAX Record Management Services (RMS), which are used to access files and their records. There are also system services that perform tasks such as resource allocation, information sharing, and input/output coordination.

Because all VAX/VMS system routines adhere to the VAX Procedure Calling Standard, you can declare any system routine as an external routine and then call the routine from a VAX PASCAL program.

VAX PASCAL supplies a source file, SYS$LIBRARY:STARLET.PAS, and an environment file, SYS$LIBRARY:STARLET.PEN, which describe every VAX/VMS system service and RMS routine. After inheriting STARLET.PEN you have access to the routines without having to declare the external routine. Section 8.6.1 describes how to inherit the system services definitions file.

In addition to the STARLET environment files, VAX PASCAL provides four VAX PASCAL source files containing condition symbol definitions such as condition values returned from a call to a system service or Run-Time Library routine. Section 8.6.2 describes how to include the symbol definitions file.

Because not all system routine declarations are provided, it is sometimes necessary for you to write your own external routine declarations. Because system routines are not written in VAX PASCAL, a VAX PASCAL program usually needs special adaptations to connect with system routines. VAX PASCAL provides many extensions to the syntax of routine declarations and calls that permit VAX/VMS system routines to be called easily from a VAX PASCAL program. Sections 8.6.3 and 8.6.5 contain detailed information on how to use the VAX PASCAL extensions to obtain the desired behavior.

8.6.1 Inheriting the System Services Definitions File

VAX PASCAL supplies a source file, SYS$LIBRARY:STARLET.PAS, and an environment file, SYS$LIBRARY:STARLET.PEN. These files contain system service routine declarations written in VAX PASCAL. Included in these files is the following information:

- External declarations of system services and RMS routines
- System symbolic names, such as system status codes and type codes for job/process information requests
- Data structures, such as record type definitions for record access blocks, file access blocks, extended attribute blocks, and name blocks

Use of the INHERIT attribute to obtain STARLET.PEN makes all VAX/VMS system services and RMS routines available to your program so that you do not have to declare them individually . For example, because the following program inherits STARLET.PEN, you do not need to define the external system service routines, thus making the program shorter and easier to maintain. As a comparison, the example in Section 8.6.5 is a similar program; however, because it does not inherit STARLET.PEN, it must explicitly declare all system routines that are used within the program.

Note that the external routine declarations in STARLET define new identifiers by which you can refer to the routines; for instance, SYS$HIBER can be referred to as $HIBER, as in the following example.

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Suspend (INPUT,OUTPUT);

TYPE
   Sys_Time = RECORD
              I,J : INTEGER;
              END;
   Unsigned_Word = [WORD] 0..65535;

VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
   Length : Unsigned_Word;
   Job_Name : VARYING[15] OF CHAR;
   Ascii_Time : VARYING[80] OF CHAR;
   Binary_Time : Sys_Time;

BEGIN
(* Print current date and time *)
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR(Current_Time, 1, Length);
(* Get name of process to suspend *)
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);

(* Get time to wake process *)
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);

(* Convert time to binary *)
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
THEN
   BEGIN
   WRITELN ('Illegal format for time string');
   HALT;
   END;

(* Suspend process *)
IF NOT ODD ($SUSPND (PRCNAM := Job_Name))
THEN
   BEGIN
   WRITELN ('Cannot suspend process');
   HALT;
   END;

(* Schedule wakeup request for self *)
IF ODD ($SCHDWK (DAYTIME := Binary_Time))
THEN
   $HIBER (* Put self to sleep *)
ELSE
   BEGIN
   WRITELN ('Cannot schedule wakeup');
   WRITELN ('Process will resume immediately');
   END;

(* Resume process *)
IF NOT ODD ($RESUME (PRCNAM := Job_Name))
THEN
   BEGIN
   WRITELN ('Cannot resume process');
   HALT;
   END;
END.
```

8.6.2 Including the Symbol Definitions Files

In addition to the STARLET.PEN environment file, VAX PASCAL provides four files containing condition symbol definitions. When you declare a system service or Run-Time Library routine, you should define the condition values by including the appropriate file in a CONST section at the beginning of your program or before the associated system service definitions. Use the %INCLUDE directive to specify the file name, as described in the *VAX PASCAL Reference Manual.*

The four symbol definition files are LIBDEF.PAS, MTHDEF.PAS, SIGDEF.PAS, and PASDEF.PAS. Note that LIBDEF.PAS, MTHDEF.PAS, and SIGDEF.PAS are provided mainly for VAX PASCAL Version 1 compatibility. These files are not updated to include new condition symbol definitions. For this reason, you should use STARLET.PEN, which contains the same definitions as the include files; however, STARLET is maintained and updated to include any new definitions.

SYS$LIBRARY:LIBDEF.PAS

This file contains definitions for all condition symbols from the general utility Run-Time Library routines. These symbols have the following form:

```
   LIB$_abc
```

For example:

```
LIB$_NOTFOU
```

SYS$LIBRARY:MTHDEF.PAS

This file contains definitions for all condition symbols from the mathematical routines library. These symbols have the following form:

```
   MTH$_abc
```

For example:

```
MTH$_SQUROONEG
```

SYS$LIBRARY:SIGDEF.PAS

This file contains miscellaneous symbol definitions used in condition handlers. These definitions are also included in STARLET.PEN. These symbols have the following form:

```
  SS$_abc
```

For example:

```
SS$_FLTOVF
```

SYS$LIBRARY:PASDEF.PAS

This file contains definitions for all condition symbols from the VAX PASCAL Run-Time Library routines. These symbols have the following form:

```
  PAS$_abc
```

For example:

```
PAS$_FILALROPE
```

8.6.3 Declaring System Routines

Before calling a routine, you must declare it. System routine names conform to one of the two following conventions:

```
  facility-code$procedure-name

  $procedure-name
```

For example, LIB$PUT_OUTPUT is the Run-Time Library routine used to write a record to the current output device and $ASCTIM is a system service routine used to convert binary time to ASCII time.

Because system routines are often called from condition handlers or asynchronous trap (AST) routines, you should declare system routines with the ASYNCHRONOUS attribute.

Each system routine is documented with a structured format in the appropriate VAX/VMS reference manual. The documentation for each routine describes the routine's purpose, the declaration format, the return value, and any required or optional parameters. Detailed information about each parameter is listed in the description. The following format is used to describe each parameter.

```
parameter-name
VMS Usage :     VMS data type
type      :     parameter data type
access    :     parameter access
mechanism :     parameter-passing mechanism
```

Using this information you must determine the parameter's data type (type), the parameter's passing semantics (access), the mechanism used to pass the parameter (mechanism) and whether the parameter is required or optional from the call format.

The following sections describe the methods available in VAX PASCAL to obtain the various data types, access methods, and passing mechanisms.

8.6.3.1 Methods Used to Obtain VMS Data Types

The data specified by a parameter has a data type. Several VMS standard data types exist. A system routine parameter must use one of these data types.

The *Introduction to VAX/VMS System Routines* lists and describes each of the VMS data types. It also includes a suggested VAX PASCAL declaration to obtain the desired data type.

8.6.3.2 Methods Used to Obtain Access Methods

The access method describes the way in which the called routine accesses the data specified by the parameter. The following three methods of access are the most common.

- Read only—data must be read by the called routine. The called routine does not write the input data. Thus, input data supplied by a variable is preserved when the called routine completes execution.
- Write only—data that the called routine returns to the calling routine must be written into a location where the calling routine can access it. Such data can be thought of as output data. The called routine does not read the contents either before or after it writes into the location.
- Modify—a parameter specifies data that is both read and written by the called routine. In this case, the called routine reads the input data, which it uses in its operations, and then overwrites the input data with the results. Thus, when the called routine completes execution, the input data specified by the argument is lost.

Table 8–7 lists all access methods that may appear under the access entry in a parameter description, as well as the VAX PASCAL translation.

Table 8–7: Access Type Translations

Access Entry	Method Used in VAX PASCAL
Call after stack unwind	Procedure or function parameter passed by immediate value.
Function call (before return)	Function parameter
Jump after unwind	Not available
Modify	Variable semantics[1]
Read only	Value or foreign semantics
Call without stack unwind	Procedure parameter
Write only	Variable semantics[1]

[1]It is possible to obtain variable semantics by either specifying the VAR keyword on the formal parameter or by passing a variable as an actual parameter using %REF, %DESCR, or %STDESCR.

8.6.3.3 Methods Used to Obtain Passing Mechanisms

The way in which an argument specifies the actual data to be used by the called routine is defined in terms of the parameter-passing mechanism. See Section 8.3 for a description of the three passing mechanisms.

There are several types of descriptors. Each descriptor contains a value that identifies the type of descriptor it is. Table 8–8 lists all passing mechanisms that may appear under the mechanism entry in a argument description and the method used in VAX PASCAL to obtain the passing mechanism.

Table 8–8: Mechanism Type Translations

Mechanism Entry	Method Used in VAX PASCAL
By value	%IMMED or [IMMEDIATE]
By reference	VAR, %REF or [REFERENCE] or default
By descriptor	
Fixed-length	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]

Table 8–8 (Cont.): Mechanism Type Translations

Mechanism Entry	Method Used in VAX PASCAL
Dynamic-string	%STDESCR parameter of type PACKED ARRAY OF CHAR or [CLASS_S]
Array	Array type, conformant array schema, or [CLASS_A]
Procedure	n.a.
Decimal-string	n.a.
Noncontiguous-array	%DESCR or [CLASS_NCA]
Varying-string	Value, VAR or %DESCR conformant parameter of type VARYING OF CHAR or %DESCR parameter of type VARYING OF CHAR
Varying-string-array	Value, VAR or %DESCR conformant parameter of type array of VARYING OF CHAR or %DESCR parameter of type array of VARYING OF CHAR
Unaligned-bit-string	n.a.
Unaligned-bit-array	n.a.
String-with-bounds	n.a.
Unaligned-bit-string-with-bounds	n.a.

Parameters passed by reference and used solely as input to a system service should be declared with Pascal value semantics; this allows actual parameters to be compile-time and run-time expressions. When a system service requires a formal parameter with a mechanism specifier, you should declare the formal parameter with the READONLY attribute to specify value semantics. Other parameters passed by reference should be declared with VAX PASCAL variable semantics to ensure that the output data is interpreted correctly. In some cases, by reference parameters are used for both input and output and should also be declared with variable semantics.

The following example shows the declaration of the Convert ASCII String to Binary Time (SYS$BINTIM) system service and a corresponding function designator. The first formal parameter requires the address of a character-string descriptor with value semantics; the second requires an address and uses variable semantics to manipulate the parameter within the service. Because you can call $BINTIM from a condition handler or AST routine, you should declare it with the ASYNCHRONOUS attribute. Also, because you may want to pass a volatile variable to the TIMADR parameter, you should use the VOLATILE attribute to indicate that the argument is allowed to be volatile.

```
TYPE
   $QUAD = [QUAD,UNSAFE] RECORD
           L0 : UNSIGNED;
           L1 : INTEGER;
           END;

VAR
   Ascii_Time : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;

[ASYNCHRONOUS,EXTERNAL(SYS$BINTIM)] FUNCTION $BINTIM
   (TIMBUF : [CLASS_S] PACKED ARRAY [$l1..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER;
   EXTERNAL;
   .
   .
   .
IF NOT ODD ($BINTIM(Ascii_Time, Binary_Time))
THEN
   BEGIN
   WRITELN ('Illegal format for time string');
   HALT;
   END;
```

8.6.3.4 Data Structure Parameters

Some system services require a parameter to be the address of a data structure that indicates a function to be performed or that holds information to be returned. Such a structure can be described as a list, a control block, or a vector. The size and POS attributes of VAX PASCAL provide an efficient method of laying out these data structures. The size attributes ensure that the fields of the data structure are of the size required by the system service, and the POS attribute allows you to position the fields correctly (see the *VAX PASCAL Reference Manual* for more information on these attributes).

For example, the Get Job/Process Information (SYS$GETJPIW) system service requires an item list consisting of an array of records of 12 bytes, where all but the last array cell requests one piece of data and the last array cell represents the item list terminator. By packing the record, you can guarantee that the fields of each record are allocated contiguously. The following program uses the system service routine $GETJPIW to retrieve the process's name as a 12-byte string.

```
[INHERIT('SYS$LIBRARY:STARLET')] PROGRAM Userid( OUTPUT );
TYPE
    Uword       = [WORD] 0..65535;
    Itmlst_Cell = PACKED RECORD
        CASE INTEGER OF
        1 : (Buf_Len   : Uword;
             Item_Code : Uword;
             Buf_Addr  : INTEGER;
             Len_Addr  : INTEGER);
        2 : (Term      : INTEGER);
        END;

VAR
    Username_String       : [VOLATILE] VARYING [12] OF CHAR;
    Itmlst                : ARRAY [1..2] OF Itmlst_Cell := ZERO;

BEGIN
Itmlst[1].Buf_Len        := 12;                               { 12 bytes returned }
Itmlst[1].Item_Code      := JPI$_USERNAME;                    { return user name  }
Itmlst[1].Buf_Addr       := IADDRESS(Username_String.BODY);   { store returned name here   }
Itmlst[1].Len_Addr       := IADDRESS(Username_String.LENGTH); { store returned length here }
Itmlst[2].Term           := 0;                                { terminate item list }

IF NOT ODD( $GETJPIW(,,,Itmlst) )
THEN WRITELN('error')
ELSE WRITELN('user name is ',Username_String);
END.
```

8.6.3.5 Default Parameters

In some cases, you do not have to supply actual parameters to correspond to all the formal parameters of a system service. In VAX PASCAL, you can supply default values for such optional parameters when you declare the service. You can then omit the corresponding actual parameters from the routine call. If you choose not to supply an optional parameter, you should initialize the formal parameter with the appropriate value, using the by immediate value (%IMMED) mechanism. Usually, the correct default value is 0.

For example, the Cancel Timer (SYS$CANTIM) system service has two optional parameters. If you do not specify values for them in the actual parameter list, you must initialize them with zeros when they are declared. The following example is the routine declaration for SYS$CANTIM.

```
[ASYNCHRONOUS,EXTERNAL(SYS$CANTIM)] FUNCTION $CANTIM
   (REQIDT : [IMMEDIATE] INTEGER := %IMMED 0;
   ACMODE : [IMMEDIATE] INTEGER := %IMMED 0)
   : INTEGER;
   EXTERNAL;
```

A call to $CANTIM must indicate the position of omitted parameters with a comma, unless they all occur at the end of the parameter list. For example, the following are legal calls to $CANTIM using the above external declaration:

```
$CANTIM (, PSL$C_USER);
$CANTIM (I);
$CANTIM;
```

PSL$C_USER is a symbolic constant that represents the value of a user access mode, and I is an integer that identifies the timer request being canceled. Note that if you call $CANTIM with both of its default parameters, you can omit the actual parameter list completely.

When it is possible for the parameter list to be truncated, you can also specify the TRUNCATE attribute on the formal parameter declaration of the optional parameter. The TRUNCATE attribute indicates that an actual parameter list for a routine can be truncated at the point that the attribute was specified. However, once one optional parameter is omitted in the actual parameter list, it is not possible to specify any optional parameter following that. For example:

```
    [EXTERNAL, ASYNCHRONOUS]
    PROCEDURE LIB$GET_FOREIGN(
      %STDESCR Get_Str      : PACKED ARRAY [l1..u1:INTEGER] OF CHAR;
      %STDESCR User_Prompt : [TRUNCATE] PACKED ARRAY [l2..u2:INTEGER] OF CHAR;
      VAR Out_Len           : [TRUNCATE] UWORD;
      VAR Force_Prompt      : [TRUNCATE] INTEGER); EXTERN;
```

With this declaration, it is not possible to specify a value for Get_Str, skip User_Prompt and specify values for Out_Len and Force_Prompt. However, it is possible to specify values for Get_Str and User_Prompt and truncate the call list at that point. In this case, two parameters would be passed in the CALL instruction.

You may want to use a combination of the %IMMED 0 and TRUNCATE methods. This combination would allow you to skip the specification of intermediate optional parameters, as well as allow you to truncate the call list once all desired parameters have been specified.

Note that VAX/VMS system services require a value to be passed by parameters, including optional parameters; therefore, you should not use the TRUNCATE attribute when defining optional parameters to a system service. Instead, you should specify default values on the formal parameter declaration.

The TRUNCATE attribute is useful when calling routines for which the optional parameter is trully optional, for example, when calling VMS Run-time library routines. For more information about the TRUNCATE attribute, see the *VAX PASCAL Reference Manual.*

8.6.3.6 Arbitrary Length Parameter Lists

Some Run-Time Library routines require a variable number of parameters. For example, there is no fixed limit on the number of values that can be passed to functions that return the minimum or maximum value from a list of input parameters. The LIST attribute supplied by VAX PASCAL allows you to indicate the mechanism by which excess actual parameters are to be passed. For example:

```
[EXTERNAL,ASYNCHRONOUS] FUNCTION MTH$DMIN1
   (Dlist : [LIST] DOUBLE)
   : DOUBLE;
   EXTERN;
```

Because the function MTH$DMIN1 returns the D_floating minimum of an arbitrary number of D_floating parameters, the formal parameter Dlist is declared with the LIST attribute. All actual parameters must be double-precision real numbers passed by reference with value semantics.

See the *VAX PASCAL Reference Manual* for details on the LIST attribute.

8.6.4 Calling System Routines

All system routines are functions that return an integer condition value; this value indicates whether the function executed successfully. An odd-numbered condition value indicates successful completion; an even-numbered condition value indicates a warning message or failure (see the *Introduction to VAX/VMS System Routines* for further explanation of how to interpret condition values). Your program can use the VAX PASCAL predeclared function ODD to test the function return value for success or failure. For example:

```
IF NOT ODD ($BINTIM(Ascii_Time,Binary_Time))
THEN
   BEGIN
     WRITELN('Illegal format for time string');
    HALT;
   END;
```

In addition, Run-Time Library routines return one or two values: the result of a computation or the routine's completion status, or both. When the routine returns a completion status, you should verify the return status before checking the result of a computation. You can use the function ODD to test for success or failure or you can check for a particular return status by comparing the return status to one of the status codes defined by the system. For example:

```
VAR
   Seed_Value : INTEGER;
   Rand_Result : REAL;

[EXTERNAL,ASYNCHRONOUS] FUNCTION MTH$RANDOM
   (VAR Seed : INTEGER)
   : REAL;
   EXTERN;
   .
   .
   .
Rand_Result := MTH$RANDOM (Seed_Value);
```

When the routine's completion status is irrelevant, your program can treat the function as though it were an external procedure and ignore the return value. For example, your program can declare the Hibernate (SYS$HIBER) system service as a function but call it as though it were a procedure:

```
[ASYNCHRONOUS,EXTERNAL(SYS$HIBER)] FUNCTION $HIBER
   : INTEGER;
   EXTERNAL;
   .
   .
   .
$HIBER; (* Put process to sleep *)
```

Because SYS$HIBER is expected to execute successfully, the program will ignore the integer condition value that is returned.

8.6.5 System Routine Example

The following example illustrates several system service calls that use VAX PASCAL extensions. The program prompts for a process name and a time string. It then suspends itself and the specified process until the correct time occurs.

```
PROGRAM Suspend (INPUT,OUTPUT);

TYPE
   $QUAD = [QUAD,UNSAFE] RECORD
           LO : UNSIGNED;
           L1 : INTEGER;
           END;
   $UWORD = [WORD] 0..65535;


VAR
   Current_Time : PACKED ARRAY[1..80] OF CHAR;
   Length : $UWORD;
   Job_Name : VARYING[15] OF CHAR;
   Ascii_Time : VARYING[80] OF CHAR;
   Binary_Time : $QUAD;

[ASYNCHRONOUS,EXTERNAL(SYS$ASCTIM)] FUNCTION $ASCTIM
   (VAR TIMLEN : [VOLATILE] $UWORD := %IMMED 0;
   %STDESCR TIMBUF : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED 0;
   TIMADR : $QUAD := %IMMED 0;
   %IMMED CVTFLG : INTEGER := %IMMED 0)
   : INTEGER;
   EXTERNAL;

[ASYNCHRONOUS,EXTERNAL(SYS$BINTIM)] FUNCTION $BINTIM
   (%STDESCR TIMBUF : PACKED ARRAY [$11..$u1:INTEGER] OF CHAR;
   VAR TIMADR : [VOLATILE] $QUAD)
   : INTEGER;
   EXTERNAL;

[ASYNCHRONOUS,EXTERNAL(SYS$HIBER)] FUNCTION $HIBER
   : INTEGER;
   EXTERNAL;

[ASYNCHRONOUS,EXTERNAL(SYS$RESUME)] FUNCTION $RESUME
   (VAR PIDADR : INTEGER := %IMMED 0;
   %STDESCR PRCNAM : PACKED ARRAY [$12..$u2:INTEGER] OF CHAR :=
      %IMMED 0)
   : INTEGER;
   EXTERNAL;
```

```
[ASYNCHRONOUS,EXTERNAL(SYS$SCHDWK)] FUNCTION $SCHDWK
   (VAR PIDADR : INTEGER := %IMMED 0;
   %STDESCR PRCNAM : PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR :=
      %IMMED 0;
   DAYTIM : $QUAD;
   REPTIM : $QUAD := %IMMED 0)
   : INTEGER;
   EXTERNAL;

[ASYNCHRONOUS,EXTERNAL(SYS$SUSPND)] FUNCTION $SUSPND
   (VAR PIDADR : INTEGER := %IMMED 0;
   %STDESCR PRCNAM : PACKED ARRAY [$l2..$u2:INTEGER] OF CHAR :=
      %IMMED 0)
   : INTEGER;
   EXTERNAL;

BEGIN
(* Print current date and time *)
$ASCTIM (TIMLEN := Length, TIMBUF := Current_Time);
WRITELN ('The current time is ', SUBSTR (Current_Time, 1,
      Length));

(* Get name of process to suspend *)
WRITE ('Enter name of process to suspend: ');
READLN (Job_Name);

(* Get time to wake process *)
WRITE ('Enter time to wake process: ');
READLN (Ascii_Time);

(* Convert time to binary *)
IF NOT ODD ($BINTIM (Ascii_Time, Binary_Time))
THEN
   BEGIN
   WRITELN ('Illegal format for time string');
   HALT;
   END;

(* Suspend process *)
IF NOT ODD ($SUSPND (PRCNAM := Job_Name))
THEN
   BEGIN
   WRITELN ('Cannot suspend process');
   HALT;
   END;
```

```
(* Schedule wakeup request for self *)
IF ODD ($SCHDWK (Daytim := Binary_Time))
THEN
   $HIBER (* Put self to sleep *)
ELSE
   BEGIN
   WRITELN ('Cannot schedule wakeup');
   WRITELN ('Process will resume immediately');
   END;

(* Resume process *)
IF NOT ODD ($RESUME (PRCNAM := Job_Name))
THEN
   BEGIN
   WRITELN ('Cannot resume process');
   HALT;
   END;
END.     (*main program *)
```

Chapter 9

Program Optimization and Efficiency

The word optimization, as used in this chapter, refers to the process of improving the efficiency of programs. The objective of optimization is to produce source and object programs that achieve the greatest amount of processing with the least amount of time and memory.

Improved program efficiency results when programs are carefully designed and written, and when compilation techniques take advantage of machine architecture. The VAX PASCAL compiler produces efficient code by utilizing the benefits of the VAX native-mode architecture and hardware. The primary goal of optimization performed by the VAX PASCAL compiler is faster program execution.

The following topics are discussed in this chapter:

- Compiler optimizations
- Programming considerations
- Optimization considerations

9.1 Compiler Optimizations

Programs compiled with the VAX PASCAL compiler undergo the process known as optimization by default. Optimization refers to a set of techniques the compiler uses to minimize the amount of time and memory required to execute a program. An optimizing compiler automatically attempts to remove repetitious instructions and redundant computations by making assumptions about the values of certain variables. This in turn reduces the size of the object code, allowing a program written in a high-level language to execute at a speed comparable to that of a well-written assembly language program. Although optimization can increase the amount of time required to compile a program, it results in a program that may execute faster and more efficiently than a nonoptimized program.

The language elements you use in the source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization (see Section 9.3). In addition, this awareness will often make it easier for you to track down the source of a problem when your program exhibits unexpected behavior.

The VAX PASCAL compiler performs the following optimizations:

- Compile-time evaluation of constant expressions
- Elimination of some common subexpressions
- Partial elimination of unreachable code
- Code hoisting from structured statements, including the removal of invariant computations from loops
- Inline code expansion for many predeclared functions
- Inline code expansion for user-declared routines
- Rearrangement of unary minus and NOT operations to eliminate unary negation and complement operations
- Partial evaluation of logical expressions
- Propagation of compile-time known values

These optimizations are described in the following sections. In addition, the VAX PASCAL compiler performs the following optimizations, which can be detected only by a careful examination of the machine code produced by the VAX PASCAL compiler.

- Global assignment of variables to registers

  If possible, reduce the number of memory references needed by assigning frequently referenced variables to registers.

- Reordering the evaluation of expressions

  This minimizes the number of temporary values required.

- Peephole optimization of instruction sequences

  The compiler examines code a few instructions at a time to find operations that can be replaced by shorter and faster equivalent operations.

9.1.1 Compile-Time Evaluation of Constants

The VAX PASCAL compiler performs the following computations on constant expressions at compile time.

- Negation of constants

  The value of a constant preceded by unary minus signs is negated at compile time. For example:

  ```
  X := -10.0;
  ```

  This is compiled as a single move instruction.

- Type conversion of constants

  The value of a lower-ranked constant is converted to its equivalent in the data type of the higher-ranked operand at compile time. For example:

  ```
  X := 10 * Y;
  ```

  If X and Y are both real numbers, then this operation is compiled as follows:

  ```
  X := 10.0 * Y;
  ```

- Arithmetic on integer and real constants

  An expression that involves +, –, *, or / operators is evaluated at compile time. For example:

```
CONST
   NN = 27;
   .
   .
   .
I := 2 * NN + J;
```

This is compiled as follows:

```
I := 54 + J;
```

- Array address calculations involving constant indexes

  These are simplified at compile time whenever possible. For example:

  ```
  VAR
     I : ARRAY[1..10,1..10] OF INTEGER;
     .
     .
     .
  I[1,2] := I[4,5];
  ```

  This is compiled as a single move instruction.
- Evaluation of constant functions and operators

  Arithmetic, ordinal, transfer, unsigned, allocation size, CARD, EXPO, and ODD functions involving constants, concatenation of string constants, and logical and relational operations on constants, are evaluated at compile time.

9.1.2 Elimination of Common Subexpressions

The same subexpression often appears in more than one computation within a program. For example:

```
A := B * C + E * F;
   .
   .
   .
H := A + G - B * C;
   .
   .
   .
IF ((B * C) - H) <> 0
THEN
   .
   .
   .
```

In this code sequence, the subexpression B * C appears three times. If the values of the operands B and C do not change between computations, the value B * C can be computed once and the result can be used in place of the subexpression. Thus, the sequence shown above is compiled as follows:

```
t := B * C;
A := t + E * F;
   .
   .
   .
H := A + G - t;
   .
   .
   .
IF ((t) - H) <> 0
THEN
   .
   .
   .
```

Two computations of B * C have been eliminated. In this case, you could have modified the source program itself for greater program optimization.

The following example shows a more significant application of this kind of compiler optimization, in which you could not reasonably modify the source code to achieve the same effect.

```
VAR
   A, B : ARRAY[1..25,1..25] OF REAL;
   .
   .
   .
A[I,J] := B[I,J];
```

Without optimization, this source program would be compiled as follows:

```
t1 := (J - 1) * 25 + I;
t2 := (J - 1) * 25 + I;
A[t1] := B[t2];
```

Variables t1 and t2 represent equivalent expressions. The VAX PASCAL compiler eliminates this redundancy by producing the following optimization:

```
t = (J - 1) * 25 + I;
A[t] := B[t];
```

9.1.3 Elimination of Unreachable Code

The VAX PASCAL compiler can determine which lines of code, if any, will never be executed and eliminates that code from the object module being produced. For example, consider the following lines from a program:

```
CONST
   Debug_Switch = FALSE;
   .
   .
   .
IF Debug_Switch
THEN
   WRITELN ('Error found here');
```

The IF-THEN statement is designed to write an error message if the value of the symbolic constant Debug_Switch is TRUE. Suppose that the error has been removed, and you change the definition of Debug_Switch to give it the value FALSE. When the program is recompiled, the compiler can determine that the THEN clause will never be executed because the IF condition is always FALSE; no machine code is generated for this clause. You need not remove the IF-THEN statement from the source program.

Note that code that is otherwise unreachable, but contains one or more labels, is not eliminated unless the GOTO statement and the label itself are located in the same block.

9.1.4 Code Hoisting from Structured Statements

The VAX PASCAL compiler can improve the execution speed and size of programs by removing invariant computations from structured statements. For example:

```
FOR J := 1 TO I + 23 DO
   BEGIN
   IF Selector
   THEN
      A[I + 23, J - 14] := 0
   ELSE
      B[I + 23, J - 14] := 1;
END;
```

If the compiler detected this IF-THEN statement, it would recognize that, regardless of the Boolean value of Selector, a value is to be stored in the array component denoted by [I + 23, J – 14]. Thus, the compiler would change the sequence to the following:

```
t := I + 23;
FOR J := 1 TO t DO
BEGIN
   u := J - 14;
   IF Selector
   THEN
      A[t,u] := 0
   ELSE
      B[t,u] := 1;
END;
```

This removes the calculation of J – 14 from the IF statement, and the calculation of I + 23 from both the IF statement and the loop.

9.1.5 Inline Code Expansion for Predeclared Functions

The VAX PASCAL compiler can often replace calls to predeclared routines with the actual algorithm for performing the calculation. For example:

```
Square := SQR (A);
```

The compiler replaces this function call with the following, and generates machine code based on the expanded call:

```
Square := A * A;
```

The program will execute faster because the algorithm for the SQR function has already been included in the machine code.

9.1.6 Inline Code Expansion for User-Declared Routines

Inline code expansion for user-declared routines performs in the same manner as inline code expansion for predeclared functions—the VAX PASCAL compiler can often replace calls to user-declared routines with an inline expansion of the routine's executable code. Inline code expansion is useful on routines that are called only a few times. The overhead of an actual procedure call is avoided; therefore, program execution is faster. The size of the program, however, may increase due to the routine's expansion.

To determine whether or not it is desirable to inline expand a routine, the compiler uses a complex heuristic. The first part of the heuristic performs tests for cases that will always prohibit the routine from being inlined. A failure of one of these tests can be thought of as a "hard failure." These "hard failure" tests verify that the following are false, if any one of these tests is true, the routine is not inlined.

- The called routine is an external or inherited routine.
- Either the calling routine or the called routine does not have inlining optimization enabled. Note that optimization is enabled by default.
- The called routine establishes an exception handler, or is used as an exception handler.
- The called function result is a structured result type.
- The calling routine and the called routine do not have the same checking options enabled.
- The calling routine and the called routine do not use the same program section.
- The called routine declares a routine parameter or is itself a routine parameter.
- The called routine's parameter list contains a LIST or TRUNCATE parameter, a read-only VAR parameter, or a conformant parameter.
- The called routine declares local file variables or contains any nonlocal GOTO operations.
- The called routine references automatic variables in an enclosing scope.

The second part of the heuristic performs tests to determine how desirable it is to inline the routine at a particular call point. A failure to one of these tests can be thought of as a "soft failure." These tests check for the number of formal parameters, number of local variables, whether the called routine is directly recursive, the number of direct calls to the routine, and the size of both the calling and the called routine.

If an explicit [OPTIMIZE(INLINE)] attribute is specified on the routine declaration, the "hard failure" tests will still be performed; however, the "soft failure" tests will not. So if the routine passes the "hard failure" tests, that routine will be inlined at all call points. Specifying this attribute provides you with more power in deciding which routines should be inlined.

NOTE

There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline routine is therefore difficult, and is not recommended.

9.1.7 Operation Rearrangement

The VAX PASCAL compiler can produce more efficient machine code by rearranging operations to avoid having to negate and then calculate the complement of the values involved. For example:

```
(-C) * (B - A)
```

If a program includes this operation, the compiler rearranges the operation to read as follows:

```
C * (A - B)
```

These two operations produce the same result, but because the compiler has eliminated negation/complement operations, the machine code produced will be more efficient.

9.1.8 Partial Evaluation of Logical Expressions

The Pascal language does not specify the order in which the components of an expression must be evaluated. If the value of an expression can be determined by partial evaluation, then some subexpressions may not be evaluated at all. This situation occurs most frequently in the evaluation of logical expressions. For example:

```
WHILE (I < 10) AND (A[I] <> 0) DO
   BEGIN
   A[I] := A[I] + 1;
   I := I + 1;
   END;
```

In this WHILE statement, the order in which the two subexpressions (I < 10) and (A[I] <> 0) are evaluated is not specified; in fact, the compiler may evaluate them simultaneously. Regardless of which subexpression is evaluated first, if its value is FALSE, the condition being tested in the WHILE statement is also FALSE. The other subexpression need not be evaluated at all. Thus, in this case, the body of the loop is never executed.

9.1.9 Value Propagation

The compiler keeps track of the values assigned to variables and traces the values to most of the places that they are used. If it is more efficient to use the value rather than a reference to the variable, the compiler makes this change. This optimization is called value propagation. Value propagation causes the object code to be smaller, and may also improve run-time speed.

Value propagation performs the following two actions:

- It allows run-time operations to be replaced with compile-time operations. For example:

  ```
  PI := 3.14;
  PIOVER2 := PI/2;
  ```

  In a program that includes these assignments, the compiler will recognize the fact that PI's value did not change between the time of PI's assignment and its use. Thus, the compiler would use PI's value instead of a reference to PI and perform the division at compile time. The compiler would treat the assignments as if they were as follows:

  ```
  PI := 3.14;
  PIOVER2 := 1.57;
  ```

  This process is repeated, allowing for further constant propagation to occur.

- It allows comparisons and branches to be avoided at run time. For example:

  ```
  X := 3;
  IF X <> 3
  THEN
     Y := 30
  ELSE
     Y := 20;
  ```

  In a program that includes these operations, the compiler would recognize that the value of X is 3 and the THEN statement cannot be reached. Thus, the compiler would generate code as if the statements were written as follows:

  ```
  X := 3;
  Y := 20;
  ```

9.1.10 Alignment of Compiler-Generated Labels

The VAX PASCAL compiler aligns the labels it generates for the top of loops and the beginnings of ELSE branches on longword boundaries, filling in unused bytes with NOP instructions. On a VAX/VMS system, a branch to a longword-aligned address is faster than a branch to an unaligned address. This optimization may increase the size of the generated code; however, it increases run-time speed.

9.1.11 Error Reduction Through Optimization

An optimized program produces results and run-time diagnostic messages identical to those produced by an equivalent unoptimized program. An optimized program may produce fewer run-time diagnostics, however, and the diagnostics may occur at different statements in the source program. For example:

Unoptimized Code	Optimized Code
A := X/Y;	t := X/Y;
B := X/Y;	A := t;
FOR I := 1 TO 10 DO	B := t;
C[I] := C[I] * X/Y;	FOR I := 1 TO 10 DO C[I] := C[I] * t;

If the value of Y is 0.0, the unoptimized program produces 12 divide-by-zero errors at run time; the optimized program produces only one. (Note that t is a temporary variable created by the compiler.) Eliminating redundant calculations and removing invariant calculations from loops can affect the detection of such arithmetic errors. You should keep this in mind when you include error-detection routines in your program.

9.2 Programming Considerations

The VAX PASCAL language elements that you use in a source program directly affect the compiler's ability to optimize the resulting object program. Therefore, you should be aware of the following ways in which you can assist compiler optimization and thus obtain a more efficient program.

- Define constant identifiers to represent values that do not change during your program. The use of constant identifiers generally makes a program easier to read, understand, and modify later. In addition, the resulting object code is more efficient because symbolic constants are evaluated only once, at compile time, while variables must be reevaluated whenever they are assigned new values.
- Whenever possible, use the structured control statements CASE, FOR, IF-THEN, IF-THEN-ELSE, REPEAT, WHILE, and WITH rather than the GOTO statement. Although the GOTO statement can be used to exit from a loop, careless use of it interferes with both optimization and the straightforward analysis of program flow.

NOTE

When both the REPEAT and WHILE statements are valid for a loop you should use the REPEAT statement because it allows for faster execution.

- Enclose in parentheses any subexpression that occurs frequently in your program. The compiler checks whether any assignments have affected the subexpression's value since its last occurrence. If the value has not changed, the compiler recognizes that a subexpression enclosed in parentheses has already been evaluated and does not repeat the evaluation. For example:

```
X := SIN (U + (B - C));
    .
    .
    .
Y := COS (V + (B - C));
```

The compiler evaluates the subexpression (B – C) as a result of performing the SIN function. When it is encountered again, the compiler checks to see whether new values have been assigned to either B or C since they were last used. If their values have not changed, the compiler does not reevaluate (B – C).

- Once your program has been completely debugged, disable all checking with either the CHECK(NONE) attribute or the /NOCHECK or /CHECK=NONE compile-time qualifier (see the *VAX PASCAL Reference Manual*). Recall that /CHECK=BOUNDS is enabled by default. When no checking code is generated, more optimizations can occur, and the program will execute faster.

  Note that integer overflow checking is disabled by default. If you are sure that your program is not in danger of integer overflow, you should not enable overflow checking. Because overflow checking precludes certain optimizations, you can achieve a more efficient program by leaving it disabled.
- When a variable is accessed by a program block other than the one in which it was declared, the variable should have static rather than automatic allocation. A variable is statically allocated if it is declared at program or module level or if it is associated with either the STATIC or PSECT attribute in a nested block. A static variable has a fixed location in memory and is therefore easy to access. An automatically allocated variable, on the other hand, has a varying location in memory; accessing it in another block is time-consuming and less efficient.
- Avoid using the same temporary variable many times in the course of a program. Instead, use a new variable every time your program needs a temporary variable. Because variables stored in registers are the easiest to access, your program is most efficient when as many variables as possible can be allocated in registers. If you use several different temporary variables, the lifetime of each one is greatly reduced; thus, there is a greater chance that storage for them can be allocated in registers rather than at memory locations.

9.3 Optimization Considerations

Because the compiler must make certain assumptions in order to optimize a program, unexpected results may occur if you do not utilize the VAX PASCAL optimizations discussed in the following sections. If your program does not execute correctly because of undesired optimizations, you can use the NOOPTIMIZE attribute or the /NOOPTIMIZE compile-time qualifier (see the *VAX PASCAL Reference Manual*) to prevent optimizations from occurring.

9.3.1 Subexpression Evaluation

An optimizing compiler, as discussed in Section 9.1, can evaluate subexpressions in any order and may even choose not to evaluate some of them. These considerations are important when the subexpressions designate functions that have side effects. For example:

```
IF F(A) AND F(B)
THEN
    .
    .
    .
```

This IF statement contains two designators for function F. If F has side effects, the compiler does not guarantee the order in which the side effects will be produced. In fact, if one call to F returns FALSE, the other call to F might never be executed, and the side effects that result from that call would never be produced.

```
Q := F(A) + F(A);
```

This expression consists of two designators for function F with the same parameter A. The Pascal standard allows a compiler to optimize the code as follows:

```
Q := 2 * F(A)
```

If the compiler does so, and function F has side effects, the side effects would occur only once because the compiler has generated code that evaluates F(A) only once.

If you disable optimization while your program is being compiled, the VAX PASCAL compiler will evaluate the expressions from left to right until the final value of the expression is determined.

9.3.2 Lowest Negative Integer

The compiler assumes that all integer values are in the range –MAXINT through MAXINT. However, the VAX architecture supports an additional integer value, (–MAXINT–1). Should your program contain a subexpression with this value, its evaluation might result in an integer overflow trap. Therefore, a computation involving the value (–MAXINT–1) might not produce the expected result. To evaluate expressions that include (–MAXINT–1), you should disable either optimization or integer overflow checking.

9.3.3 Pointer References

The compiler assumes that the value of a pointer variable is either the constant identifier NIL or a reference to a variable allocated in heap storage by the NEW procedure. A variable allocated in heap storage is not declared in a VAR section and has no identifier of its own; you can refer to it only by the name of a pointer variable, followed by a circumflex (^).

```
VAR
    X : INTEGER;
    P : ^INTEGER;

BEGIN
  NEW (P);
  P^ := 0;
  X  := 0;
IF P^ = X
THEN
  P^ := P^ + 1;
END.
```

If a pointer variable in your program must refer to a variable with an explicit name, that variable must be declared VOLATILE or READONLY. The compiler makes no assumptions about the value of volatile variables and therefore performs no optimizations on them (the VOLATILE attribute is fully described in the *VAX PASCAL Reference Manual*).

Use of the ADDRESS function, which creates a pointer to a variable, can result in a warning message because of optimization characteristics. By passing a non-read-only or nonvolatile static or automatic variable as the parameter to the ADDRESS function, you indicate to the compiler that the variable was not allocated by NEW, but was declared with its own identifier. Because the compiler's assumptions are incorrect, a warning message occurs (the ADDRESS function is fully described in the *VAX PASCAL Reference Manual*). Note that you can also use IADDRESS, which functions similarly to the ADDRESS function, except that IADDRESS does not generate any warning messages. You should use caution when using IADDRESS—see the *VAX PASCAL Reference Manual*) for more information.

Similarly, when the parameter to ADDRESS is a formal VAR parameter or a component of a formal VAR parameter, the compiler issues a warning message that not all dynamic variables allocated by NEW may be passed to the function.

9.3.4 Variant Records

Because all the variants of a record variable are stored in the same memory location, a program can use several different field identifiers to refer to the same storage space. However, only one variant is valid at a given time; all other variants are undefined. Thus, you must store a value in a field of a particular variant before you attempt to use it. For example:

```
VAR
   X : INTEGER;
   A : RECORD
        CASE T : BOOLEAN OF
        TRUE   : (B : INTEGER);
        FALSE  : (C : REAL);
        END;
   .
   .
   .
X := A.B + 5;
A.C := 3.0;
X := A.B + 5;
```

Record A has two variants, B and C, which are located at the same storage address. When the assignment A.C := 3.0 is executed, the value of A.B becomes undefined because TRUE is no longer the currently valid variant. When the statement X := A.B + 5 is executed for the second time, the value of A.B is unknown. The compiler may choose not to evaluate A.B a second time because it has retained the field's previous value. To eliminate any misinterpretations caused by this assumption, variable A should be associated with the VOLATILE attribute. The compiler makes no assumptions about the value of VOLATILE objects (see the *VAX PASCAL Reference Manual*).

9.3.5 Type Cast Operations

When a type cast operation is performed, the compiler disregards any previous assumptions about the value of the cast object. This allows you to temporarily alter the type of the cast object at that point only. A type cast operation affects optimization only at the location in the program where the cast takes place. Optimizations elsewhere in the program and optimizations involving only uncast objects are not affected.

You should use type casts with care because the type cast operation can sometimes affect distant parts of a program. If a type cast on a variable is likely to affect its value at other points in the program, the variable should be declared VOLATILE (see the *VAX PASCAL Reference Manual*).

9.3.6 Effects of Optimization on Debugging

When you compile a VAX PASCAL program, the resulting object code is optimized by default. The compiler automatically allocates variables in registers, removes invariant expressions that occur within loops so that they are evaluated outside the loop, and so on.

Some of the effects of optimized programs on debugging are as follows:

- Use of registers

  When the VAX PASCAL compiler determines that the value of an expression does not change between two given occurrences, it may save the value in a register. In such a case, it does not recompute the value for the next occurrence, but assumes that the value saved in the register is valid. If, while debugging the program, you use the DEPOSIT command to change the value of the variable in the expression, then the value of that variable is changed, but the corresponding value stored in the register is not. Thus, when execution continues, the value in the register may be used instead of the changed value in the expression, causing unexpected results.

  When the value of a variable is being held in a register, its value in memory is generally invalid; therefore, a spurious value may be displayed if the EXAMINE command is issued for a variable under these circumstances.

- Coding order

  Some of the compiler optimizations cause code to be generated in a order different from the way it appears in the source. Sometimes code is eliminated altogether. This causes unexpected behavior when you use the VAX/VMS Debugger to step by line or use the source display features.

- Use of condition codes

  This optimization technique takes advantage of the way in which the VAX processor condition codes are set. For example, consider the following source code:

  ```
  X := X + 2.5;
  IF X < 0
  THEN
        .
        .
        .
  ```

  Rather than test the new value of X to determine whether to branch, the optimized object code bases its decision on the condition code settings after 2.5 is added to X. Thus, if you attempt to set a debugging breakpoint at the second line and deposit a different value into X, you will not achieve the intended result because the condition codes no longer reflect the value of X. In other words, the decision to branch is being made without regard to the deposited value of the variable.

- Inline code expansion on user-declared routines

  There is no stack frame for an inline user-declared routine and no debugger symbol table information for the expanded routine. Debugging the execution of an inline user-declared routine is therefore difficult, and is not recommended.

To prevent conflicts between optimization and debugging, you should always compile your program with the /NOOPTIMIZE qualifier until it is thoroughly debugged. Then you can recompile the program (which by default will be optimized) to produce efficient code.

Appendix A

Diagnostic Messages

This appendix summarizes the error messages that can be generated by a VAX PASCAL program at compile time and at run-time.

A.1 Compiler Diagnostics

The VAX PASCAL compiler reports compile-time diagnostics in the source listing (if one is being generated) 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:

$$\texttt{\%PASCAL-}\left\{\begin{array}{c}\texttt{I-}\\ \texttt{W-}\\ \texttt{E-}\\ \texttt{F-}\end{array}\right\}$$

- I indicates an informational message that flags VAX extensions to the Pascal standard, identifies unused or possibly uninitialized variables, or provides additional information about a more severe error.
- W indicates a warning that flags an error that may cause unexpected results, but that does not prevent the program from linking and executing.
- E indicates an error that prevents generation of machine code; instead, the compiler produces an empty object module indicating that E-level messages were detected in the source program.
- F indicates a fatal error.

If the source program contains either E- or F-level messages, the errors must be corrected before the program can be linked and executed.

All diagnostic messages have explanatory text. These messages and their severity levels are listed as follows. For those messages that are not self-explanatory, a brief explanation of the error is provided. Portions of the message text enclosed in quotation marks are items that the compiler substitutes with the name of a data object when it generates the message.

ABSALIGNCON, Absolute address / alignment conflict

> **Error.** The address specified by the AT attribute does not have the number of low-order bits implied by the specified alignment attribute.

ACCMETHCON, Specified ACCESS_METHOD conflicts with file's record organization

> **Warning.** You cannot specify ACCESS_METHOD:=DIRECT for a file that has indexed organization or sequential organization and variable-length records. You cannot specify ACCESS_METHOD:=KEYED for a file with sequential or relative organization.

ACTHASNOFRML, Actual parameter has no corresponding formal parameter

> **Error.** The number of actual parameters specified in a routine call exceeds the number of formal parameters in the routine's declaration, and the last formal parameter does not have the LIST attribute.

ACTMULTPL, Actual parameter specified more than once

> **Error.** Each formal parameter (except one with the LIST attribute) can have only one corresponding actual parameter.

ACTPASCNVTMP, Conversion: actual passed is resulting temporary
ACTPASRDTMP, Formal requires read access: actual parameter is resulting temporary
ACTPASSIZTMP, Size mismatch: actual passed is resulting temporary
ACTPASWRTMP, Formal requires write access: actual parameter is resulting temporary

> **Warning.** A temporary variable is created if an actual parameter does not have the size, type, and accessibility properties required by the corresponding foreign formal parameter.

ACTPRMORD, Actual parameter must be ordinal

Error. The actual parameter that specifies the starting index of an array for the PACK or UNPACK procedure must have an ordinal type.

ADDIWRDALIGN, ADD_INTERLOCKED requires variable with at least word alignment
ADDIWRDSIZE, ADD_INTERLOCKED requires 16-bit variable

Error. These restrictions are imposed by the VAX ADAWI instruction.

ADDRESSVAR, "parameter name" is a VAR parameter, ADDRESS is illegal

Warning. You should not use the ADDRESS function on a non volatile variable or component or on a formal VAR parameter.

ALIGNAUTO, Alignment greater than 2 conflicts with automatic allocation

Error. The VAX hardware aligns the stack on a longword boundary; therefore, you cannot specify a greater alignment for automatically allocated variables.

ALIGNFNCRES, Alignment greater than 2 not allowed on function result

Error. The use of an attribute on a routine conflicts with the requirements of the object's type.

ALIGNINT, ALIGNED expression must be INTEGER value in range 0..9

Error.

ALIGNVALPRM, Alignment greater than 2 not allowed on value parameter

Error. The use of an attribute on a parameter conflicts with the requirements of the object's type.

ALLPRMSAM, All parameters to 'MIN' or 'MAX' must have the same type

Error.

APARMACTDEF, Anonymous parameter "parameter number" has neither actual nor default

Error. If the declaration of a routine failed to specify a name for a formal parameter, a call to the routine will result in this error message. Note that the routine declaration would also cause an error to be reported.

ARITHOPNDREQ, Arithmetic operand(s) required

Error.

ARRCNTPCK, Array cannot be PACKED

Error. At least one parameter to the PACK or UNPACK procedure must be unpacked.

ARRHAVSIZ, "routine name" requires that ARRAY component have compile-time known size

Error. You cannot use the PACK and UNPACK procedures to pack or unpack one multidimensional conformant array into another. The component type of the dimension being copied must have a compile-time known size; that is, it must have some type other than a conformant schema.

ARRMSTPCK, Array must be PACKED

Error. At least one parameter to the PACK or UNPACK procedure must be of type PACKED.

ARRNOTSTR, Array type is not a string type

Error. You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. The only legal array, therefore, is PACKED ARRAY [1..n] OF CHAR.

ASYREQASY, ASYNCHRONOUS "calling routine" requires that "called routine" also be ASYNCHRONOUS

Warning.

ASYREQVOL, ASYNCHRONOUS "routine name" requires that "variable name" be VOLATILE

Warning. A variable referred to in a nested asynchronous routine must have the VOLATILE attribute.

ATINTUNS, AT address must be an INTEGER or UNSIGNED value

Error.

ATREXTERN, "attribute name" attribute allowed only on external routines

Error. The LIST and CLASS_S attributes can be specified only with the declarations of external routines.

ATTRCONCMDLNE, Attribute contradicts command line qualifier

Error. The double-precision attribute specified contradicts the /G_FLOATING or /NOG_FLOATING qualifier specified with the PASCAL command.

ATTRCONFLICT, Attribute conflict: "attribute name"

Information. This message can appear as additional information on other error messages.

ATTRONTYP, Descriptor class attribute not allowed on this type

Error. The use of the descriptor class attribute on the variable, parameter, or routine conflicts with the requirements of the object's type.

AUTOGTRMAXINT, Allocation of "variable name" causes automatic storage to exceed MAXINT bits

Error. The VAX implementation restricts automatic storage to a size of 2,147,483,647 bits.

BADSETCMP, < and > not permitted in set comparisons

Error.

BINOCTHEX, Expecting BIN, OCT, or HEX

Error. You must supply BIN, OCT, or HEX as a variable modifier when reading the variable on a non-decimal basis.

BLKNOTFND, "routine" block "routine name" declared FORWARD in "block name" is missing

Error.

BLKTOODEEP, Routine blocks nested too deeply

Error. You cannot nest more than 31 routine blocks.

BNDACTDIFF, Actual's array bounds differ from those of other parameters in same section

Error. All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible.

BNDCNFRUN, Bounds of conformant ARRAY "array name" not known until run-time

Error. You cannot use the UPPER and LOWER functions on a dynamic array parameter in a compile-time constant expression.

BNDSUBORD, Bound expressions in a subrange type must be ordinal

Error. The expressions that designate the upper and lower limits of a subrange must be of an ordinal type.

BOOLOPREQ, BOOLEAN operand(s) required

Error. The operation being performed requires operands of type BOOLEAN. Such operations include the AND, OR, and NOT operators and the SET_INTERLOCKED and CLEAR_INTERLOCKED functions.

BOOSETREQ, BOOLEAN or SET operand(s) required

Error.

BYTEALIGN, Type larger than 32 bits can be positioned only on a byte boundary

Error. See Chapter 5 for information on the types that are allocated more than 32 bits.

CALLFUNC, Function "function name" called as procedure, function value discarded

Warning.

CARCONMNGLS, CARRIAGE_CONTROL parameter is meaningless given file's type

Warning. The carriage control parameter is usually meaningful only for files of type TEXT and VARYING OF CHAR.

CASLABEXPR, Case label and case selector expressions are not compatible

Error. All case labels in a CASE statement must be compatible with the expression specified as the case selector.

CASORDRELPTR, Compile-time cast allowed only between ordinal, real, and pointer types
CASSELORD, Case selector expression must be an ordinal type

Error.

CASSRCSIZ, Source type of a cast must have a size known at compile-time
CASTARSIZ, Target type of a cast must have a size known at compile-time

Error. A variable being cast by the type cast operator cannot be a conformant array or a conformant VARYING parameter. An expression being cast cannot be a conformant array parameter, a conformant VARYING parameter, or a VARYING OF CHAR expression. The target type of the cast cannot be VARYING OF CHAR.

CDDABORT, %DICTIONARY processing of CDD record definition aborted

Error. The VAX PASCAL compiler is unable to process the CDD record description. See the accompanying CDD messages for more information.

CDDBADDIR, %DICTIONARY directive not allowed in deepest %INCLUDE, ignored

Error. A program cannot use the %DICTIONARY directive in the fifth nested %INCLUDE level. The compiler ignores all %DICTIONARY directives in the fifth nested %INCLUDE level.

CDDBADPTR, invalid pointer was specified in CDD record description

Warning. The CDD pointer data type refers to a CDD path name that cannot be extracted, and is replaced by ^INTEGER.

CDDBIT, Ignoring bit field in CDD record description

Information. The VAX PASCAL compiler cannot translate a CDD bit data type that is not aligned on a byte boundary and whose size is greater than 32 bits.

CDDBLNKZERO, Ignoring blank when zero attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD BLANK WHEN ZERO clause.

CDDCOLMAJOR, CDD description specifies a column-major array

Error. The VAX PASCAL compiler supports only row-major arrays. Change the CDD description to specify a row-major array.

CDDDEPITEM, Ignoring depends item attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD DEPENDING ON ITEM attribute.

CDDDFLOAT, D_Floating CDD datatype was specified when compiling with G_FLOATING

Warning. The CDD record description contains a D_floating data type while compiling with G_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDFLDVAR, CDD record description contains field(s) after CDD variant clause

Error. The CDD record description contains fields after the CDD variant clause. Because VAX PASCAL translates a CDD variant clause into a Pascal variant clause, and a Pascal variant clause must be the last field in a record type definition, the fields following the CDD variant clause are illegal.

CDDGFLOAT, G_Floating CDD datatype was specified when compiling with NOG_FLOATING

Warning. The CDD record description contains a G_floating data type while compiling with D_floating enabled. It is replaced with [BYTE(8)] RECORD END.

CDDILLARR, Aligned array elements can not be represented, replacing with [BIT(n)] RECORD END

Information. The VAX PASCAL compiler does not support CDD record descriptions that specify an array whose array elements are aligned on a boundary greater than the size needed to represent the data type. It is replaced with [BIT(n)] RECORD END, where n is the appropriate length in bits.

CDDINITVAL, Ignoring specified initial value specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD INITIAL VALUE clause.

CDDMINOCC, Ignoring minimum occurs attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD MINIMUM OCCURS attribute.

CDDONLYTYP, %DICTIONARY may only appear in a TYPE definition part

Error. The %DICTIONARY directive is allowed only in the TYPE section of a program.

CDDRGHTJUST, Ignoring right justified attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD JUSTIFIED RIGHT clause.

CDDSCALE, Ignoring scaled attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD scaled data types.

CDDSRCTYPE, Ignoring source type attribute specified in CDD record description

Information. The VAX PASCAL compiler does not support the CDD source type attribute.

CDDTAGDEEP, CDD description nested variants too deep

Error. A CDD record description may not include more than 15 levels of CDD variants. The compiler ignores variants beyond the fifteenth level.

CDDTAGVAR, Ignoring tag variable and any tag values specified in CDD record description

Information. The VAX PASCAL compiler does not fully support CDD VARIANTS OF field description statement. The specified tag variable and any tag values are ignored.

CDDTOODEEP, CDD description nested too deep

Error. Attributes for the CDD record description exceed implementation's limit for record complexity. Modify the CDD description to reduce the level of nesting in the record description.

CDDTRUNCREF, Reference string which exceeds 255 characters has been truncated

Information. The VAX PASCAL compiler does not support reference strings greater than 255 characters.

CDDUNSTYP, Unsupported CDD datatype "standard data type name"

Information. The CDD record description for an item has attempted to use a data type that is not supported by VAX PASCAL. The VAX PASCAL compiler makes the data type accessible by declaring it as [BYTE(n)] RECORD END where n is the appropriate length in bytes. Change the data type to one that is supported by VAX PASCAL or manipulate the contents of the field by passing it to external routines as variables or by using the VAX PASCAL type casting capabilities to perform an assignment.

CLSCNFVAL, CLASS_S is only valid with conformant strings passed by value

Error. When the CLASS_S attribute is used in the declaration of an internal routine, it can be used only on a conformant PACKED ARRAY OF CHAR. The conformant variable must also be passed by value semantics.

CLSNOTALLW, "descriptor class name" not allowed on a parameter of this type

Error. Descriptor class attributes are not allowed on formal parameters defined with either an immediate or a reference passing mechanism.

CMTBEFEOF, Comment not terminated before end of input

Error.

CNFCANTCNF, Component of PACKED conformant schema cannot be conformant

Error.

CNTBEARRCMP, Not allowed on an array component
CNTBEARRIDX, Not allowed on an array index
CNTBECAST, Not allowed on a cast
CNTBECNFCMP, Not allowed on a conformant array component
CNTBECNFIDX, Not allowed on a conformant array index
CNTBECNFVRY, Not allowed on a conformant varying component
CNTBECOMP, Not allowed on a compilation unit
CNTBECONST, Not allowed on a CONST definition part
CNTBEDEFDECL, Not allowed on any definition or declaration part
CNTBEDESPARM, Not allowed on a %DESCR foreign mechanism parameter
CNTBEEXESEC, Not allowed on an executable section
CNTBEFILCMP, Not allowed on a file component
CNTBEFUNC, Not allowed on a function result
CNTBEIMMPARM, Not allowed on a parameter passed by an immediate passing mechanism
CNTBELABEL, Not allowed on a LABEL declaration part
CNTBEPCKCNF, Not allowed on a PACKED conformant array component

CNTBEPTRBAS, Not allowed on a pointer base
CNTBERECFLD, Not allowed on a record field
CNTBEREFPARM, Not allowed on a parameter passed by a reference passing mechanism
CNTBERTNDECL, Not allowed on a routine declaration
CNTBERTNPARM, Not allowed on a routine parameter
CNTBESETRNG, Not allowed on a set range
CNTBESTDPARM, Not allowed on a %STDESCR foreign mechanism parameter
CNTBETAGFLD, Not allowed on a variant tag field
CNTBETAGTYP, Not allowed on a variant tag type
CNTBETYPDEF, Not allowed on a type definition
CNTBETYPE, Not allowed on a TYPE definition part
CNTBEVALPARM, Not allowed on a value parameter
CNTBEVALUE, Not allowed on a VALUE initialization part
CNTBEVALVAR, Not allowed on a VALUE variable
CNTBEVAR, Not allowed on a VAR declaration part
CNTBEVARBLE, Not allowed on a variable
CNTBEVARPARM, Not allowed on a VAR parameter
CNTBEVRYCMP, Not allowed on a varying component

Information. These messages can appear as additional information on other error messages.

COMCONFLICT, COMMON "block name" conflicts with another COMMON or PSECT of same name

Error. You can allocate only one variable in a particular common block, and the name of the common block cannot be the same as the names of other common blocks or program sections used by your program.

CSTRBADTYP, Constructor: only ARRAY, RECORD, or SET type
CSTRCOMISS, Constructor: component(s) missing
CSTRNOVRNT, Constructor: no matching variant
CSTRREFAARR, Repetition factor allowed only in ARRAY constructors
CSTRREFAINT, Repetition factor must be INTEGER
CSTRREFALRG, Repetition factor too large
CSTRREFANEG, Repetition factor cannot be negative
CSTRTOOMANY, Constructor: too many components

Error. You can write constructors only for data items of an ARRAY type. You must specify one and only one value in the constructor for each component of the type. In an array constructor, you cannot use a negative integer value as a repetition factor to specify values for consecutive components.

CTGARRDESC, Contiguous array descriptor cannot describe size/alignment properties

Information. Conformant array parameters, dynamic array parameters, and %DESCR array parameters all use the contiguous array descriptor mechanism in the VAX Procedure Calling Standard. Size and alignment attributes are prohibited on such arrays, as these attributes can create noncontiguous allocation. This message can appear as additional information in other error messages.

DEBUGOPT, /NOOPTIMIZE is recommended with /DEBUG

Information. Unexpected results may be seen when debugging an optimized program. To prevent conflicts between optimization and debugging, you should compile your program with /NOOPTIMIZE until it is thoroughly debugged. Then you can recompile the program with optimization enabled to produce more efficient code.

DEFRTNPARM, Default parameter syntax not allowed on routine parameters
DEFVARPARM, Default parameter syntax not allowed on VAR parameters

Error.

DESCTYPCON, Descriptor class / type conflict

> **Error.** The descriptor class for parameter passing conflicts with the parameter's type. Refer to Section 8.3.3 for legal descriptor class/type combinations.

DIRCONVISIB, Directive contradicts visibility attribute

> **Error.** The EXTERN, EXTERNAL, and FORTRAN directives conflict directly with the LOCAL and GLOBAL attributes.

DONTPACKVAR, "routine name" is illegal, variable can never appear in a packed context

> **Error.** You cannot call the BITSIZE and BITNEXT functions for conformant parameters.

DUPLALIGN, Alignment already specified
DUPLALLOC, Allocation already specified
DUPLATTR, Attribute already specified
DUPLCLASS, Descriptor class already specified
DUPLDOUBLE, Double precision already specified

> **Error.** Only one member of a particular attribute class can appear in the same attribute list.

DUPLGBLNAM, Duplicated global name

> **Warning.** The GLOBAL attribute cannot appear on more than one variable or routine with the same name.

DUPLMECH, Passing mechanism already specified
DUPLOPT, Optimization already specified
DUPLSIZE, Size already specified
DUPLVISIB, Visibility already specified

> **Error.** Only one member of a particular attribute class can appear in the same attribute list.

DUPTYPALI, Alignment already specified by type identifier "type name"
DUPTYPALL, Allocation already specified by type identifier "type name"
DUPTYPATR, Attribute already specified by type identifier "type name"
DUPTYPDES, Descriptor class already specified by type identifier "type name"
DUPTYPSIZ, Size already specified by type identifier "type name"
DUPTYPVIS, Type identifier "type name" already carries a visibility attribute

Error. An attribute specified for an object was already specified in the definition of the object's type.

ELEOUTRNG, Element out of range

Error. A value specified in a set constructor used as a compile-time constant expression does not fall within the subrange defined as the set's base type.

EMPTYCASE, Empty case body

Error. You failed to specify any case labels and corresponding statements in the body of a CASE statement.

ENVERROR, Environment resulted from a compilation with Errors

Error. The environment file inherited by the program compiled with errors. Unexpected results may occur in the program now being compiled.

ENVFATAL, Environment resulted from a compilation with Fatal Errors

Error. The environment file inherited by the program compiled with fatal errors. Unexpected results may occur in the program now being compiled.

ENVOLDVER, Environment was created by a VAX PASCAL V2 compiler, please recompile

Warning. The environment file inherited by the program was created by a VAX PASCAL V2 compiler. You should regenerate the environment file with the VAX PASCAL V3 compiler.

ENVWARN, Environment resulted from a compilation with Warnings

Warning. The environment file inherited by the program compiled with warnings. Unexpected results may occur in the program now being compiled.

ERREALCNST, Error in real constant: digit expected

Error.

ERRNONPOS, ERROR parameter can be specified only with nonpositional syntax

Error.

ERRORLIMIT, Error Limit = "current error limit", source analysis terminated

Fatal. The error limit specified for the program's compilation was exceeded; the compiler was unable to continue processing the program. By default, the error limit is set at 30, but you can use the /ERROR_LIMIT qualifier at compile time to change it.

ESTBASYNCH, ESTABLISH requires that "routine name" be ASYNCHRONOUS

Warning.

EXPLCONVREQ, Explicit conversion to lower type required

Error. An expression of a higher-ranked type cannot be assigned to a variable of a lower-ranked type; you must first convert the higher-ranked expression by using DBLE, SNGL, TRUNC, ROUND, UTRUNC, or UROUND, as appropriate.

EXPRARITH, Expression must be arithmetic

Error. An expression whose type is not arithmetic cannot be assigned to a variable of a real type.

EXPRARRIDX, Expression is incompatible with unpacked array's index type

Error. The index type of the UNPACKed array is not compatible with the index type of either the PACK or UNPACK procedure it was passed to.

EXPRCOMTAG, Expression is not compatible with tag type

Error. A case label specified for a NEW, DISPOSE, or SIZE routine must be assignment compatible with the tag type of the variant.

EXPRNOTSET, Expression is not a SET type

Error. The compiler encountered an expression of some type other than SET in a call to the CARD function.

EXTRNALLOC, Allocation attribute conflicts with EXTERNAL visibility

Error. The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

EXTRNAMDIFF, External names are different

Information. This message can appear as additional information on other error messages.

EXTRNCFLCT, "PSECT or FORWARD" conflicts with EXTERNAL visibility

Error. The storage for an external variable or routine is not allocated by the current compilation; therefore, the specification of an allocation attribute is meaningless.

FILEVALASS, FILE evaluation / assignment is not allowed

Error. You cannot attempt to evaluate a file variable or assign values to it.

FILOPNDREQ, FILE operand required

Error. The EOF, EOLN, and UFB functions require parameters of file types.

FILVARFIL, FILE_VARIABLE parameter must be of a FILE type

Error. The file variable parameter to the OPEN and CLOSE procedures must denote a file variable.

FLDIVPOS, Field "field name" is illegally positioned

Error. A POS attribute attempted to position a record field before the end of the previous field in the declaration.

FLDONLYTXT, Field width allowed only when writing to a TEXT file
FLDWDTHINT, Field-width expression must be of type INTEGER

Error.

FORACTORD, FOR loop control variable must be of an ordinal type
FORACTVAR, FOR loop control must be a true variable

> **Error.** The control variable of a FOR statement must be a simple variable of an ordinal type and must be declared in a VAR section. For example, it cannot be a field in a record that was specified by a WITH statement, or a function identifier.

FORCTLVAR, "variable name" is a FOR control variable

> **Warning.** The control variable of a FOR statement cannot be assigned a value; used as a parameter to the ADDRESS function; passed as a writeable VAR, %REF, %DESCR, or %STDESCR parameter; used as the control variable of a nested FOR statement; or written into by a READ, READLN, or READV procedure.

FORINEXPR, Expression is incompatible with FOR loop control variable

> **Error.** The type of the initial or final value specified in a FOR statement is variable.

FRMLPRMDESC, Formal parameters use different descriptor formats
FRMLPRMINCMP, Formal routine parameters are not compatible
FRMLPRMNAM, Formal parameters have different names
FRMLPRMSIZ, Formal parameters have different size attributes
FRMLPRMTYP, Formal parameters have different types

> **Information.** These messages can appear as additional information on other error messages.

FRSTPRMSTR, READV requires first parameter to be a string expression

> **Error.** You must specify at least two parameters for the READV procedure—a character-string expression and a variable into which new values will be read.

FRSTPRMVARY, WRITEV requires first parameter to be a variable of type VARYING

> **Error.**

FUNCTRESTYP, Routine must be declared as FUNCTION to specify a result type

> **Error.** You cannot specify a result type on a PROCEDURE declaration.

FUNRESTYP, Function result types are different

> **Information.** This message can appear as additional information on other error messages.

FWDREPATRLST, Declared FORWARD; repetition of attribute list not allowed
FWDREPPRMLST, Declared FORWARD; repetition of formal parameter list not allowed
FWDREPRESTYP, Declared FORWARD; repetition of result type not allowed

> **Error.** If the heading of a routine has the FORWARD directive, the declaration of the routine body cannot repeat the formal parameter list, the result type (applies only if the routine is a function), or any attribute lists that appeared in the heading.

FWDWASFUNC, FORWARD declaration was FUNCTION
FWDWASPROC, FORWARD declaration was PROCEDURE

> **Error.**

GOTONOTALL, GOTO not allowed to jump into a structured statement

> **Warning.**

GTR32BITS, "routine name" cannot accept parameters larger than 32 bits

> **Error.** DEC and UDEC cannot translate objects larger than 32 bits into their textual equivalent.

HIDATOUTER, HIDDEN legal only on definitions and declarations at outermost level

> **Error.** When an environment file is being generated, it is possible to prevent information concerning a declaration from being included in the environment file by using the HIDDEN attribute. However, because an environment file consists only of declarations and definitions at the outermost level of a compilation unit, the HIDDEN attribute is legal only on these definitions and declarations.

IDENTGTR31, Identifier longer than 31 characters exceeds capacity of compiler

> **Warning.**

IDXNOTCOMPAT, Index type is not compatible with declaration

Error. The type of an index expression is not assignment compatible with the index type specified in the array's type definition.

IDXNOTLAB, Identifier is not declared as a label

Error. The identifier specified in the GOTO statement is declared as a type other than LABEL.

IDXREQDKEY, Creating INDEXED organization requires dense keys

Warning. When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's alternate keys must be dense; that is, you may not omit any key numbers in the range from 0 through the highest key number specified for the file's component type.

IDXREQKEY0, Creating INDEXED organization requires FILE OF RECORD with at least KEY(0)

Warning. When you specify ORGANIZATION:=INDEXED when opening a file with HISTORY := NEW or UNKNOWN, the file's component type must be a record for which a primary key, designated by the [KEY(0)] attribute, is defined.

IMMEDBNDROU, Immediate passing mechanism may not be used on bound routine "routine name"

Warning. You cannot prefix a formal or an actual routine parameter with the immediate passing mechanism unless the routine was declared with the UNBOUND attribute.

IMMEDUNBND, Routines passed by immediate passing mechanism must be UNBOUND

Warning. A formal routine parameter that has the immediate passing mechanism must also have the UNBOUND attribute.

IMMGTR32, Immediate passing mechanism not allowed on values larger than 32 bits

Error. See Chapter 5 for more information on the types that are allocated more than 32 bits.

IMMHAVSIZ, Type passed by immediate passing mechanism must have compile-time known size

Error. You cannot specify an immediate passing mechanism for a conformant parameter or a formal parameter of type VARYING OF CHAR.

INCMPBASE, Incompatible with SET base type

Error. If no type identifier denotes the base type of a set constructor, the first element of the constructor determines the base type. The type of all subsequent elements specified in the constructor must be compatible with the type of the first.

INCMPOPND, Incompatible operand(s)

Error. The types of one or more operands in an expression are not compatible with the operation being performed.

INCMPPARM, Incompatible "routine" parameter

Error. An actual routine parameter is incompatible with the corresponding formal parameter.

INCMPTAGTYP, Incompatible variant tag types

Error. This message can appear as additional information on other error messages.

INCTOODEEP, %INCLUDE directives nested too deeply, ignored

Error. A program cannot include more than five levels of files with the %INCLUDE directive. The compiler ignores %INCLUDE files beyond the fifth level.

INDNOTORD, Index type must be an ordinal type

Error. The index type of an array must be an ordinal type.

INITNOEXT, INITIALIZE routine may not be EXTERNAL
INITNOFRML, INITIALIZE routine must have no formal parameter list

Error.

INPNOTDECL, INPUT not declared in heading

Error. A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

INVCASERNG, Invalid range in case label list

Error.

INVEVAL, Array or Record evaluation not allowed

Error.

IVATTR, Unrecognized attribute

Error.

IVAUTOMOD, AUTOMATIC variable is illegal at the outermost level of a MODULE

Error. You cannot specify the AUTOMATIC attribute for a variable declared at module level.

IVCHKOPT, Unrecognized CHECK option

Warning.

IVCOMBFLOAT, Illegal combination of D_floating and G_floating

Error. You cannot combine D_floating and G_floating numbers in a binary operation.

IVDIRECTIVE, Unrecognized directive

Error. The directive following a procedure or function heading is not one of those recognized by the VAX PASCAL compiler.

IVENVIRON, Environment "environment name" has illegal format, source analysis terminated

Fatal. The environment file inherited by the program has an illegal format; compilation is immediately aborted. However, a listing will still be produced if one was being generated.

IVFUNC, Invalid use of function "function name"
IVFUNCALL, Invalid use of function call
IVFUNCID, Invalid use of function identifier

Error. These messages result from illegal attempts to assign values or otherwise refer to the components of the function result (if its type is structured), use the type cast operator on a function identifier or its result, or deallocate the storage reserved for the function result (if its type is a pointer).

IVKEYOPT, Unrecognized KEY option

Error.

IVKEYVAL, FINDK KEY_VALUE cannot be an array (other than PACKED ARRAY [1..n] OF CHAR)

Error.

IVKEYWORD, Missing or unrecognized keyword

Error. The compiler failed to find an identifier where it expected one in a call to the OPEN or CLOSE procedure, or it found an identifier that was not legal in this position in the parameter list.

IVMATCHTYP, Invalid MATCH_TYPE parameter to FINDK

Error.

IVOPTMOPT, Unrecognized OPTIMIZE option

Warning.

IVQUALFILE, Illegal qualifier "qualifier name" on file specification

Warning. Only the /LIST and /NOLIST qualifiers are allowed on the file specification of a %INCLUDE directive.

IVQUOCHAR, Illegal nonprinting character (ASCII "nnn") within quotes

Warning. The only nonprinting characters allowed in a quoted string are the space and tab; the use of other nonprinting characters in a string causes this warning. To include nonprinting characters in a string, you should use the extended string syntax described in the *VAX PASCAL Reference Manual.*

IVRADIXDGIT, Illegal digit in binary, octal, or hexadecimal constant

Error.

IVREDECL, Illegal redeclaration gives "symbol name" multiple meanings in "scope name"
IVREDECLREC, Illegal redeclaration gives "symbol name" multiple meanings in this record
IVREDEF, Illegal redefinition gives "symbol name" multiple meanings in "scope name"

Warning. When an identifier is used in any given block, it must have the same meaning wherever it appears in the block.

IVUSEALIGN, Invalid use of alignment attribute
IVUSEALLOC, Invalid use of allocation attribute

Error.

IVUSEATTR, Invalid use of "attribute name" attribute

Error. The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEATTRLST, Invalid use of an attribute list

Error.

IVUSEBNDID, Illegal use of bound identifier "identifier name"

Error. An identifier that represents one bound of a conformant schema was used where a variable was expected, such as in an assignment statement or in a formal VAR parameter section. The restrictions on the use of a bound identifier are identical to those on a constant identifier.

IVUSEDES, Invalid use of descriptor class attribute

Error. The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

IVUSEFNID, Illegal use of function identifier "identifier name"

Error. Two examples of illegal uses are the assignment of values to the components of the function result (if its type is structured) and the passing of the function identifier as a VAR parameter.

IVUSESIZ, Invalid use of size attribute

> **Error.** The use of an attribute on a variable, parameter, or routine conflicts with the requirements of the object's type.

KEYINTRNG, KEY number must be an INTEGER value in range 0..254

> **Error.** The key number specified by a KEY attribute must fall in the integer subrange 0..254.

KEYNOTALIGN, KEY "key number" field "field name" at bit position "bit position" is unaligned
KEYORDSTR, KEY allowed only on ordinal and fixed-length string fields
KEYPCKREC, KEY field in PACKED RECORD must have an alignment attribute
KEYREDECL, Key number "key number" is multiply defined
KEYSIZ1_2_4, Size of an ordinal key must be 1, 2 or 4 bytes
KEYSIZ2_4, Size of a signed integer key must be 2 or 4 bytes
KEYSIZSTR, Size of a string key cannot exceed 255 bytes
KEYUNALIGN, KEY field cannot be UNALIGNED

> **Error.**

LABDECIMAL, Label number must be expressed in decimal radix

> **Error.**

LABINCTAG, Variant case label's type is incompatible with tag type

> **Error.** The type of a constant specified as a case label of a variant record is not assignment compatible with the type of the tag field.

LABNOTFND, No definition of label "label name" in statement part of "block name"

> **Error.** A label that you declared in a LABEL section does not prefix a statement in the executable section.

LABREDECL, Redefinition of label "label name" in "block name"

> **Error.** A label cannot prefix more than one statement in the same block.

LABRNGTAG, Variant case label does not fall within range of tag type

Error. A constant specified as a case label of a variant record is not within the range defined for the type of the tag field.

LABTOOBIG, Label "label number" is greater than MAXINT

Error.

LABUNDECL, Undeclared label "label name"

Error. VAX PASCAL requires that you declare all labels in a LABEL declaration section before you use them in the executable section.

LABUNSATDECL, Unsatisfied declaration of label "label name" is not local to "block name"

Error. A label that prefixes a statement in a nested block was declared in an enclosing block.

LIBESTAB, LIB$ESTABLISH is incompatible with VAX PASCAL; use predeclared procedure ESTABLISH

Warning. VAX PASCAL establishes its own condition handler for processing PASCAL-specific run-time signals. Calling LIB$ESTABLISH directly replaces the handler supplied by the compiler with a user-written handler; the probable result is improper handling of run-time signals. You should use PASCAL's predeclared ESTABLISH procedure to establish user-written condition handlers.

LISTONEND, LIST attribute allowed only on final formal parameter

Error.

LISTUSEARG, Formal parameter has LIST attribute, use predeclared function ARGUMENT

Error. A formal parameter with the LIST attribute cannot be directly referenced. You should use the predeclared function ARGUMENT to reference the actual parameters corresponding to the formal parameter.

LNETOOLNG, Line too long, is truncated to 255 characters

Error. A source line cannot exceed 255 characters. If it does, the compiler disregards the remainder of the line.

LOWGTRHIGH, Low-bound exceeds high-bound

Error. The definition of the flagged subrange type is illegal because the value specified for the lower limit exceeds that for the upper limit.

MAXLENINT, Max-length must be a value of type INTEGER

Error. The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MAXLENRNG, Max-length must be in range 1..65535

Error. The maximum length specified for type VARYING OF CHAR must be an integer in the range 1..65535; that is, the type definition must denote a legal character string.

MECHEXTERN, Foreign mechanism specifier allowed only on external routines

Error.

MISSINGEND, No matching END, expected near line "line number"

Information. The compiler expected an END statement at a location where none was found. Compilation proceeds as though the END statement were correctly located.

MODOFNEGNUM, MOD of a negative modulus has no mathematical definition

Error. In the MOD operation A MOD B, the operand B must have a positive integer value. This message is issued only when the MOD operation occurs in a compile-time constant expression.

MSTBEARRAY, Type must be ARRAY

Error.

MSTBEARRVRY, Type must be ARRAY or VARYING

Error. You cannot use the syntax [index] to refer to an object that is not of type ARRAY or VARYING OF CHAR.

MSTBEBOOL, Control expression must be of type BOOLEAN

Error. The IF, REPEAT, and WHILE statements require a Boolean control expression.

MSTBEREC, Type must be RECORD

Error.

MSTBERECVRY, Type must be RECORD or VARYING

Error. You cannot use the syntax "Variable.Identifier" to refer to an object that is not of type RECORD or VARYING OF CHAR.

MSTBESTAT, Cannot initialize non-STATIC variables

Error. You cannot initialize variables declared without the STATIC attribute in nested blocks, nor can you initialize program-level variables whose attributes give them some allocation other than static.

MSTBETEXT, "I/O routine" requires FILE_VARIABLE of type TEXT

Error. The READLN and WRITELN procedures operate only on text files.

MULTDECL, "symbol name" has multiple conflicting declarations, reason(s):

Error.

NCATOA, Cannot reformat content of actual's CLASS_NCA descriptor as CLASS_A

Error. This message can appear as additional information on other error messages.

NEWQUADAGN, "type name"'s base type is ALIGNED("nnn"); NEW handles at most ALIGNED(3)

Error. You cannot call the NEW procedure to allocate pointer variables whose base types specify alignment greater than a quadword. To allocate such variables, you must use external routines.

NOASSTOFNC, Block does not contain an assignment to function result "function name"

Warning. The block of a function must include a statement that assigns a return value to the function identifier.

NODECLVAR, "symbol name" is not declared in a VAR section of "block name"

Error. You cannot initialize a variable using the VALUE section if the variable was not declared in the same block in which the VALUE section appears.

NODSCREC, No descriptor class for RECORD type

Error. The VAX Procedure Calling Standard does not define a descriptor format for records; therefore, you cannot specify %DESCR for a parameter of type RECORD.

NOFLDREC, No field "field name" in RECORD type "type name"

Error. The field specified does not exist in the specified record.

NOFRMINDECL, Declaration of "routine" parameter "routine name" supplied no formal parameter list

Information. You specified actual parameters in a call on a formal routine parameter that was declared with no formal parameters. Although such a call was legal in VAX PASCAL Version 1, it does not follow the rules of the Pascal standard. You should edit your program to reflect this change.

NOINITEXT, Initialization not allowed on EXTERNAL variables
NOINITINH, Initialization not allowed on inherited variables

Error. You can initialize only those variables whose storage is allocated in this compilation.

NOINITVAR, Cannot initialize "symbol name"—it is not declared as a variable

Error. Variables are the only data items that can be initialized, and they can be initialized only once.

NOLISTATTR, Parameter to this predeclared function must have LIST attribute

Error. ARGUMENT and ARGUMENT_LIST_LENGTH require their first parameter to be a formal parameter with the LIST attribute.

NOREPRE, No textual representation for values of this type

Error. You cannot write a value to a text file (using WRITE or WRITELN) or to a VARYING string (using WRITEV) if there is no textual representation for the type. Similarly, you cannot read a value from a text file (using READ or READLN) or from a VARYING string (using READV) if there is no textual representation for the type. Such types are RECORD, ARRAY (other than PACKED ARRAY [1..n] OF CHAR), SET, and pointer.

NOTAFUNC, "symbol name" is not declared as a "routine."

Error. An identifier followed by a left parenthesis, a semicolon, or one of the reserved words END, UNTIL, and ELSE is interpreted as a call to a routine with no parameters. This message is issued if the identifier was not declared as a procedure or function identifier. Note that in the current version, functions can be called with the procedure call statement.

NOTASYNCH, "routine name" is not ASYNCHRONOUS

Information. This message can appear as additional information on other error messages.

NOTATYPE, "symbol name" is not a type identifier

Error. An identifier that does not represent a type was used in a context where the compiler expected a type identifier.

NOTAVAR, "symbol name" is not declared as a variable

Error. You cannot assign a value to any object other than a variable.

NOTAVARFNID, "symbol name" is not declared as a variable or a function identifier

Error. You cannot assign a value to any object other than a variable or a function identifier.

NOTBEADDR, May not be parameter to ADDRESS
NOTBEASSIGN, May not be assigned
NOTBECALL, May not be called as a FUNCTION
NOTBECAST, May not be type cast
NOTBEDEREF, May not be dereferenced
NOTBEDES, May not be passed by untyped %DESCR
NOTBEEVAL, May not be evaluated
NOTBEFILOP, May not be used in a file operation
NOTBEFLD, May not be field selected
NOTBEFNCPRM, May not be passed as a FUNCTION parameter
NOTBEFORCTL, May not be used as FOR loop variable
NOTBEFORDES, May not be passed as a descriptor foreign parameter
NOTBEFOREF, May not be passed as a reference foreign parameter
NOTBEIADDR, May not be parameter to IADDRESS
NOTBEIDX, May not be indexed
NOTBEIMMED, May not be passed by untyped immediate passing mechanism
NOTBENEW, May not be written into be NEW
NOTBENSTCTL, May not be control variable for an inner FOR loop
NOTBEREAD, May not be written into be READ
NOTBEREF, May not be passed by untyped reference passing mechanism
NOTBERODES, May not be passed as a READONLY descriptor foreign parameter
NOTBEROFOR, May not be passed as a READONLY reference foreign parameter
NOTBEROVAR, May not be passed as a READONLY VAR parameter
NOTBETOUCH, May not be read/modified/written

NOTBEVAR, May not be passed as a VAR parameter
NOTBEWODES, May not be passed as a WRITEONLY descriptor foreign parameter
NOTBEWOFOR, May not be passed as a WRITEONLY reference foreign parameter
NOTBEWOVAR, May not be passed as a WRITEONLY VAR parameter
NOTBEWRTV, May not be parameter to WRITEV

Information. These messages can appear as additional information on other error messages.

NOTBYTOFF, Field "field name" is not aligned on a byte boundary

Error.

NOTDECLROU, "symbol name" is not declared as a "routine."
NOTINITIAL, "routine name" is not INITIALIZE

Information. These messages can appear as additional information on other error messages.

NOTINRNG, Value does not fall within range of the tag type

Error. The value specified as the case label of a variant record is not a legal value of the tag field's type. This message is also issued if a case label in a call to NEW, DISPOSE, or SIZE falls outside the range of the tag type.

NOTSAMTYP, Not the same type
NOTUNBOUND, "routine name" is not UNBOUND

Information. These messages can appear as additional information on other error messages.

NOTVARNAM, Parameter to this predeclared function must be simple variable name

Error. The parameter cannot be indexed, be dereferenced, have a field selected, or be an expression. It must be the name of the entire variable.

NOTVOLATILE, "variable name" is non-VOLATILE

Warning. You should not use the ADDRESS function on a non volatile variable or component or on a formal VAR parameter.

NOUNSATDECL, No unsatisfied declaration of label "label name" in "block name"

Error.

NUMFRMLPARM, Different numbers of formal parameters

Information. This message can appear as additional information on other error messages.

NXTACTDIFF, NEXT of actual's component differs from that of other parameters in same section

Error. All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size and alignment of the array's inner dimensions.

OLDDECLSYN, Obsolete "routine" parameter declaration syntax

Information. The declaration of a formal routine parameter uses the obsolete Version 1 syntax. You should edit your program to incorporate the current version syntax, which is mandated by the Pascal standard.

OPNDASSCOM, Operands are not assignment compatible
OPNOTINT, Operand(s) must be of type INTEGER or UNSIGNED

Error.

OPTREQV46, Use of a KEY attribute option requires VMS Version 4.6

Information.

ORDOPNDREQ, Ordinal operand(s) required

Error or Warning. This message is at warning level if you try to use INT, ORD, or UINT on a pointer expression. It is at error level if you use PRED or SUCC on an expression whose type is not ordinal.

OUTNOTDECL, OUTPUT not declared in heading

Error. A call to EOF, EOLN, READLN, or WRITELN did not specify a file variable, and the default INPUT or OUTPUT was not listed in the program heading.

OVRDIVZERO, Overflow or division by zero in compile-time expression

Error.

PACKSTRUCT, "component name" of a PACKED structured type

Error or Warning. You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writeable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

PARMACTDEF, Formal parameter "parameter name" has neither actual nor default

Error. If a formal parameter is not declared with a default, you must pass an actual parameter to it when calling its routine.

PARMCLAMAT, Parameter section classes do not match

Information. This message can appear as additional information on other error messages.

PARMLIMIT, VAX architectural limit of 255 parameters exceeded

Error. You cannot declare a procedure with more than 255 formal parameters. A function whose result type requires that the result be stored in more than 64 bits or whose result type is a character string cannot have more than 254 formal parameters. In a call to a routine declared with the LIST attribute, you also cannot pass more than 255 (or 254) actual parameters.

PARMSECTMAT, Division into parameter sections does not match

Information. This message can appear as additional information on other error messages.

PASPREILL, Passing predeclared "routine name" is illegal

Error. You cannot use the IADDRESS function on a predeclared routine for which there is no corresponding routine in the VAX/VMS Run-Time Library (such as the interlocked functions). In addition, you cannot pass a predeclared routine as a parameter if there is no way to write the predeclared routine's formal parameter list in VAX PASCAL. Examples of the latter case are the PRED and SUCC functions and many of the I/O routines.

PASSEXTERN, Passing mechanism allowed only on external routines

Error.

PCKARRBOO, PACKED ARRAY OF BOOLEAN parameter expected

Error.

PCKUNPCKCON, Packed/unpacked conflict

Information. This message can appear as additional information on other error messages.

POSAFTNONPOS, Positional parameter cannot follow a nonpositional parameter

Error.

POSALIGNCON, Position / alignment conflict

Error. The bit position specified by the POS attribute does not have the number of low-order bits implied by the specified alignment attribute.

POSINT, POS expression must be a positive INTEGER value

Error.

PREREQPRMLST, Passing predeclared "routine name" requires formal to include parameter list

Error. To pass one of the predeclared routines EXPO, ROUND, TRUNC, UNDEFINED, UTRUNC, UROUND, DBLE, SNGL, QUAD, INT, ORD, and UINT as an actual parameter to a routine, you must specify a formal parameter list in the corresponding formal routine parameter.

PRMKWNSIZ, Parameter must have a size known at compile-time

Error. The BIN, HEX, OCT, DEC, and UDEC functions cannot be used on conformant parameters. The SIZE and NEXT functions cannot be used on conformant parameters in compile-time constant expressions.

PROCESSFILE, Compiling file "file name"

Information.

PROCESSRTN, Generating code for routine "routine name"

Information.

PROPRMEXT, Declaration of "program parameter name" is EXTERNAL—program parameter files must be locally allocated
PROPRMFIL, A program parameter must be a variable of type FILE
PROPRMINH, Declaration of "program parameter name" is inherited—program parameter files must be locally allocated
PROPRMLEV, Program parameter "program parameter name" is not declared as a variable at the outermost level

Error. Any external file variable (other than INPUT and OUTPUT) that is listed in the program heading must also be declared as a file variable in a VAR section in the program block.

PSECTMAXINT, Allocation of "symbol name" causes PSECT "PSECT name" to exceed MAXINT bits

Error. The VAX implementation restricts the size of a program section to 2,147,483,647 bits.

PTRCMPEQL, Pointer values may only be compared for equality

Error. The equality (=) and inequality (<>) operators are the only operators allowed for values of a pointer type; all other operators are illegal.

PTREXPRCOM, Pointer expressions are not compatible

Error. The base types of two pointer expressions being compared for equality (=) or inequality (<>) are not structurally compatible.

QUOBEFEOL, Quoted string not terminated before end of line

Error.

QUOSTRING, Quoted string expected

Error. The compiler expects the %DICTIONARY and %INCLUDE directives, and the radix notations for binary (%B), hexadecimal (%X), and octal constants (%O), to be followed by a quoted string of characters.

RADIXTEXT, Radix input requires FILE_VARIABLE of type TEXT

Error. The input radix specifiers (BIN, OCT, and HEX) operate only on text files.

READONLY, "variable name" is READONLY

Warning. You cannot use a read-only variable in any context that would store a new value in the variable. For example, a read-only variable cannot be used in a file operation.

REALCNSTRNG, Real constant out of range

Error. See the *VAX PASCAL Reference Manual* for details on the range of real numbers.

REALOPNDREQ, Real (SINGLE, DOUBLE or QUADRUPLE) operand(s) required

Error.

RECHASFILE, Record contains one or more FILE components, POS is illegal

Error.

RECLENINT, RECORD_LENGTH expression must be of type INTEGER

Error. The value of the record length parameter to the OPEN procedure must be an integer.

RECLENMNGLS, RECORD_LENGTH parameter is meaningless given file's type

Warning. The record length parameter is usually relevant only for files of type TEXT and VARYING OF CHAR.

RECMATCHTYP, MATCH_TYPE identifier "NXT or NXTEQL" is recommended instead of "GTR or GEQ"

Information.

REDECL, A declaration of "symbol name" already exists in "block name"

Error. You cannot redeclare an identifier or a label in the same block in which it was declared. Inheriting an environment is equivalent to including all of its declarations at program or module level.

REDECLATTR, "attribute name" already specified

Error. Only one member of a particular attribute class can appear in the same attribute list.

REDECLFLD, Record already contains a field "field name"

Error. The names of the fields in a record must be unique; they cannot be duplicated between variants.

REINITVAR, "variable name" has already been initialized

Error. Variables are the only data items that can be initialized, and they can be initialized only once.

REPCASLAB, Value has already appeared as a label in this CASE statement

Error. You cannot specify the same value more than once as a case label in a CASE statement.

REPFACZERO, Repetition factor cannot be the function ZERO
REQCLAORNCA, Arrays and conformants of this parameter type require either CLASS_A or CLASS_NCA
REQCLS, Scalars and strings of this parameter type require CLASS_S

Error.

REQNOCH, Primary key requires NOCHANGES option

Error.

REQPKDARR, The combination of CLASS_S and %STDESCR requires a PACKED ARRAY OF CHAR structure

Error.

REQREADVAR, READ or READV requires at least one variable to read into

Error. The READ and READV procedures require that you specify at least one variable to be read from a file.

REQWRITELEM, WRITE requires at least one write-list-element

Error. The WRITE procedure requires that you specify at least one item to be written to a file.

REVRNTLAB, Value has already appeared as a label in this variant part

Error. You cannot specify the same value more than once as a case label in a variant part of a record.

RTNSTDESCR, Routines cannot be passed using %STDESCR

Error.

SENDSPR, Internal Compiler Error

Fatal. An error has occurred in the execution of the VAX PASCAL compiler. Along with this message, you will receive information that helps you find the location in the source program and the name of the compilation phase at which the error occurred. You may be able to rewrite the section of your program that caused the error and thus successfully compile the program. However, even if you are able to remedy the problem, please submit a Software Performance Report (SPR) to DIGITAL and provide a machine-readable copy of the program.

SEQ11FORT, PDP-11 specific directive SEQ11 treated as equivalent to FORTRAN directive

Information.

SETBASCOM, SET base types are not compatible

Error. The base type of two sets used in a set operation are not compatible.

SETELEORD, SET element expression must be of an ordinal type

Error. The expressions used to denote the elements of a set constructor or the bounds of a set type definition must have an ordinal type.

SETNOTRNG, SET element is not in range 0..255

Error. In a set whose base type is a subrange of integers or unsigned integers, all set elements in the set's type definition or in a constructor for the set must be in the range 0..255.

SIZACTDIFF, SIZE of actual differs from that of other parameters in same section

Error. All actual parameters passed to a formal parameter section whose type is a conformant schema must have identical bounds and be structurally compatible. This message refers to the allocation size of the array's outermost dimension.

SIZARRNCA, Explicit size on ARRAY dimension makes CLASS_NCA mandatory

Error.

SIZATRTYPCON, Size attribute / type conflict

Error. For an ordinal type, the size specified must be at least as large as the packed size but no larger than 32 bits. Pointer types and type SINGLE must be allocated exactly 32 bits, type DOUBLE exactly 64 bits, and type QUADRUPLE exactly 128 bits. For types ARRAY, RECORD, SET, and VARYING OF CHAR, the size specified must be at least as large as their packed sizes. For the details of allocation sizes in VAX PASCAL, see Chapter 5.

SIZCASTYP, Variable's size conflicts with cast's target type

Error. In a type cast operation, the size of the variable and the size of the type to which it is cast must be identical.

SIZEDIFF, Sizes are different

Information. This message can appear as additional information on other error messages.

SIZEINT, Size expression must be a positive INTEGER value

Error.

SIZGTRMAX, Size exceeds MAXINT bits

Error. The size of a record or an array type or the size speci\-fied by a size attribute exceeds 2,147,483,647 bits. The VAX implementation imposes this size restriction.

SIZMULTBYT, Size of component of array passed by descriptor is not a multiple of bytes

Error. When an array or a conformant parameter is passed using the %DESCR mechanism specifier, the descriptor built by the compiler must follow the VAX Procedure Calling Standard. Such a descriptor can describe only an array whose components fall on byte boundaries.

SPEOVRDECL, Foreign mechanism specifier required to override parameter declaration

Error. When you specify a default value for a formal VAR or routine parameter, you must also use a mechanism specifier to override the characteristics of the parameter section.

SPURIOUS, "error message" at "line number"—"column number"

Information. The compiler did not correctly note the location of this error in your program and later could not position and print the correct error message. You may be able to correct the section of your program that caused the error and thus avoid this error. Please submit a Software Performance Report (SPR) and provide a machine-readable copy of the program if you receive this error.

SRCERRORS, Source errors inhibit continued compilation—correct and recompile

Fatal. A serious error previously detected in the source program has corrupted the compiler's symbol tables and inhibits further compilation. Correct the serious error and recompile the program.

SRCTXTIGNRD, Source text following end of compilation unit ignored

Warning. The compiler ignores any text following the END statement that terminates a compilation unit. This error probably resulted from an unmatched END statement in your program.

STDACTINCMP, Nonstandard: actual is not name compatible with other parameters in same section

Information. According to the Pascal standard, all actual parameters passed to a parameter section must have the same type identifier or the same type definition. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDATTRLST, Nonstandard: attribute list
STDBIGLABEL, Nonstandard: label number greater than 9999
STDBLANKPAD, Nonstandard: blank-padding used during string operation
STDCALLFUNC, Nonstandard: function "function name" called as a procedure
STDCASLBLRNG, Nonstandard: label range in case selector
STDCAST, Nonstandard: type cast operator
STDCMPCOMPAT, Nonstandard: cannot "PACK or UNPACK", array component types are incompatible
STDCOMPDIR, Nonstandard: compiler directive
STDCNFARR, Nonstandard: conformant array syntax
STDCONCAT, Nonstandard: concatenation operator

Information. These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCONST, Nonstandard: "type name" constant

Information. Binary, hexadecimal, and octal constants and constants of type DOUBLE, QUADRUPLE, and UNSIGNED are VAX extensions to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDCTLDECL, Nonstandard: control variable "variable name" not declared in VAR section of "block name"

Information. The Pascal standard requires that the control variable of a FOR statement be declared in the same block in which the FOR statement appears.

STDDECLSEC, Nonstandard: declaration sections either out of order or duplicated in "block name"

Information. In the Pascal standard, the declaration sections must appear in the order LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION. The ability to specify the sections in any order is a VAX extension. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDEFPARM, Nonstandard: default parameter declaration

Information. This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDDIRECT, Nonstandard: "directive name" directive

Information. The EXTERN, EXTERNAL, FORTRAN, and SEQ11 directives are VAX extensions to Pascal. (FORWARD is the only directive specified by the Pascal standard.) This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPCASLST, Nonstandard: empty case-list element

Information. This message is issued if you do not specify any case labels and executable statements between two semicolons or between OF and a semicolon in the CASE statement. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDEMPPARM, Nonstandard: empty actual parameter position

Information. This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPREC, Nonstandard: empty record section

Information. The Pascal standard does not allow record type definitions of the form RECORD END;. This message appears only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPSTR, Nonstandard: empty string

Information. This message refers to VAX extensions to Pascal and is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDEMPVRNT, Nonstandard: empty variant

Information. This message occurs if you do not specify a variant between two semicolons or between OF and a semicolon. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDERRPARM, Nonstandard: error-recovery parameter
STDEXPON, Nonstandard: exponentiation operator
STDEXTSTR, Nonstandard: extended string syntax
STDFORMECH, Nonstandard: foreign mechanism specifier

Information. These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDFLDHIDPTR, Nonstandard: record field identifier "field identifier name" hides type identifier "field identifier name"

Information.

STDFUNCTRES, Nonstandard: FUNCTION returning a value of a "type name" type

Information. The ability of functions to have structured result types is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDINCLUDE, Nonstandard: %INCLUDE directive
STDINITVAR, Nonstandard: initialization syntax in VAR section

Information. These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDMATCHVRNT, Nonstandard: no matching variant label

Information. This message is issued if you call the NEW or DISPOSE procedure, and one of the case labels specified in the call does not correspond to a case label in the record variable. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDMODCTL, Nonstandard: potential uplevel modification of "variable name" prohibits use as control variable

Information. You cannot use as the control variable of a FOR statement any variable that might be modified in a nested block. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDMODULE, Nonstandard: MODULE declaration

> **Information.** The item listed in this message is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNILCON, Nonstandard: use of reserved word NIL as a constant

> **Information.** Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, and constants of BOOLEAN and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDNOFRML, Nonstandard: FUNCTION or PROCEDURE parameter declaration lacks formal parameter list

> **Information.** This message is issued if you try to pass actual parameters to a formal routine parameter for which you declared no formal parameter list. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDNONPOS, Nonstandard: nonpositional parameter syntax
STDOTHER, Nonstandard: OTHERWISE clause
STDPASSPRE, Nonstandard: passing predeclared "routine name"

> **Information.** These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPCKSET, Nonstandard: combination of packed and unpacked sets

> **Information.** The Pascal standard does not allow packed and unpacked sets to be combined in set operations. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDPREDECL, Nonstandard: predeclared "routine"

> **Information.** Many predeclared procedures and functions are VAX extensions to Pascal. The use of these routines causes this message to be issued if you have specified the /STANDARD qualifier with the PASCAL command.

STDPRETYP, Nonstandard: predefined type "type name"

Information. The types SINGLE, DOUBLE, QUADRUPLE, UNSIGNED, and VARYING OF CHAR are VAX extensions to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDRADIX, Nonstandard: radix constant

Information. The item listed in this message is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDRDBIN, Nonstandard: binary input from a TEXT file
STDRDENUM, Nonstandard: enumerated type input from a TEXT file
STDRDHEX, Nonstandard: hexadecimal input from a TEXT file
STDRDOCT, Nonstandard: octal input from a TEXT file
STDRDSTR, Nonstandard: string input from a TEXT file

Information. The Pascal standard allows only INTEGER, CHAR, and REAL values to be read from a text file. The ability to read values of other types is a VAX extension to Pascal. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDREDECLNIL, Nonstandard: redeclaration of reserved word NIL

Information. The Pascal standard considers NIL a reserved word, while VAX PASCAL considers it to be a predeclared identifier. Thus, if you have specified the /STANDARD qualifier with the PASCAL command, this message will be issued if you attempt to redefine NIL.

STDREM, Nonstandard: REM operator

Information. The item listed in this message is a VAX extension to Pascal. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSIMCON, Nonstandard: only simple constant (optional sign) or quoted string

Information. Only simple constants and quoted strings are allowed by the Pascal standard to appear as constants. Simple constants are integers, character strings, real constants, symbolic constants, constants of type BOOLEAN, and enumerated types. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSPECHAR, Nonstandard: "$" or "_" in identifier
STDSTRCOMPAT, Nonstandard: string compatibility

Information. These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSTRUCT, Nonstandard: types do not have same name

Information. Because the Pascal standard does not recognize structural compatibility, two types must have the same type identifier or type definition to be compatible. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDSYMLABEL, Nonstandard: symbolic label

Information. These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDTAGFLD, Nonstandard: invalid use of tag field

Information. The tag field of a variant record cannot be a parameter to the ADDRESS function, nor can you pass it as a writeable VAR, %REF, %DESCR, or %STDESCR formal parameter. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDUNSAFE, Nonstandard: UNSAFE compatibility

Information. If you have used the UNSAFE attribute on an object that is later tested for compatibility, you will receive this message. You must also have specified the /STANDARD qualifier with the PASCAL command.

STDUSEDCNF, Nonstandard: conformant array used as a string
STDUSEDPCK, Nonstandard: PACKED ARRAY [1..1] OF CHAR used as a string
STDVALUE, Nonstandard: VALUE initialization section
STDVAXCDD, Nonstandard: %DICTIONARY directive

> **Information.** These messages refer to VAX extensions to Pascal and are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVALCNFPRM, Nonstandard: conformant array may not be passed to value conformant parameter

> **Information.**

STDVRNTPART, Nonstandard: empty variant part

> **Information.** According to the Pascal standard, a variant part that declares no case labels and field lists between the words OF and END is illegal. This message occurs only if you have specified the /STANDARD qualifier with the PASCAL command.

STDVRNTRNG, Nonstandard: variant labels do not cover the range of the tag type

> **Information.** According to the Pascal standard, you must specify one case label for each value in the tag type of a variant record. This message is issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STDWRTBIN, Nonstandard: binary output to a TEXT file
STDWRTENUM, Nonstandard: user defined enumerated type output to a TEXT file
STDWRTHEX, Nonstandard: hexadecimal output to a TEXT file
STDWRTOCT, Nonstandard: octal output to a TEXT file

> **Information.** The Pascal standard allows only INTEGER, BOOLEAN, CHAR, REAL, and PACKED ARRAY [1..n] OF CHAR values to be written to a text file. The ability to write values of other types is a VAX extension to Pascal. These messages are issued only if you have specified the /STANDARD qualifier with the PASCAL command.

STREQLLEN, String values must be of equal length

> **Error.** You cannot perform string comparisons on character strings that have different lengths.

STROPNDREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) operand required
STRPARMREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) parameter required
STRTYPREQ, String (CHAR, PACKED ARRAY [1..n] OF CHAR, or VARYING) type required

Error. The file name parameter to the OPEN procedure and the parameter to the LENGTH function must be character strings of the types listed.

SYNASCII, Illegal ASCII character
SYNASSERP, Syntax: ":=", ";" or ")" expected
SYNASSIGN, Syntax: ":=" expected
SYNASSSEMI, Syntax: ":=" or ";" expected
SYNATRCAST, Syntax: attribute list not allowed on a type cast
SYNATTTYPE, Syntax: attribute-list or type specification
SYNBEGDECL, Syntax: BEGIN or declaration expected
SYNBEGIN, Syntax: BEGIN expected
SYNCOASSERP, Syntax: ",", ":=", ";" or ")" expected
SYNCOELRB, Syntax: ",", ".." or "]" expected
SYNCOLCOMRP, Syntax: ":", "," or ")" expected
SYNCOLON, Syntax: ":" expected
SYNCOMCOL, Syntax: "," or ":" expected
SYNCOMDO, Syntax: "," or DO expected
SYNCOMMA, Syntax: "," expected
SYNCOMRB, Syntax: "," or "]" expected
SYNCOMRP, Syntax: "," or ")" expected
SYNCOMSEM, Syntax: "," or ";" expected
SYNCONTMESS, Syntax: CONTINUE or MESSAGE expected
SYNCOSERP, Syntax: ",", ";" or ")" expected
SYNDIRBLK, Syntax: directive or block expected

Error. The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNDIRMIS, Syntax: directive missing, EXTERNAL assumed

Error. In the absence of a directive where one is expected, the compiler assumes that EXTERNAL is the intended directive and proceeds with compilation based on that assumption.

SYNDO, Syntax: DO expected
SYNELIPSIS, Syntax: ".." expected
SYNELSESTMT, Syntax: ELSE or start of new statement expected
SYNEND, Syntax: END expected
SYNEQL, Syntax: "=" expected
SYNERRCTE, Error in compile-time expression
SYNEXPR, Syntax: expression expected
SYNEXSEOTEN, Syntax: expression, ";", OTHERWISE or END expected
SYNFUNPRO, Syntax: FUNCTION or PROCEDURE expected
SYNHEADTYP, Syntax: routine heading or type identifier expected
SYNIDCAEND, Syntax: identifier, CASE or END expected
SYNIDCARP, Syntax: identifier, CASE or ")" expected
SYNIDCASE, Syntax: identifier or CASE expected
SYNIDENT, Syntax: identifier expected
SYNILLEXPR, Syntax: ill-formed expression
SYNINT, Syntax: integer expected
SYNINVSEP, Syntax: invalid token separator
SYNIVATRLST, Syntax: illegal attribute list
SYNIVPARM, Syntax: illegal actual parameter
SYNIVPRMLST, Syntax: illegal actual parameter list
SYNIVSYM, Syntax: illegal symbol
SYNIVVAR, Syntax: illegal variable
SYNLABEL, Syntax: label expected
SYNLBRAC, Syntax: "[" expected
SYNLPAREN, Syntax: "(" expected
SYNLPASEM, Syntax: "(" or ";" expected
SYNLPCORB, Syntax: "(", "," or "]" expected
SYNLPSECO, Syntax: "(", ";" or ":" expected
SYNMECHEXPR, Syntax: mechanism specifier or expression expected
SYNNEWSTMT, Syntax: start of new statement expected
SYNOF, Syntax: OF expected
SYNPARMLST, Syntax: actual parameter list

SYNPARMSEC, Syntax: parameter section expected
SYNPERIOD Syntax: "." expected.
SYNPROMOD, Syntax: PROGRAM or MODULE expected
SYNQUOSTR, Syntax: quoted string expected
SYNRBRAC, Syntax: "]" expected
SYNRESWRD, Syntax: reserved word cannot be redefined
SYNRPAREN, Syntax: ")" expected
SYNRPASEM, Syntax: ";" or ")" expected
SYNRTNTYPCNF, Syntax: routine heading, type identifier or conformant schema expected
SYNSEMI, Syntax: ";" expected
SYNSEMIEND, Syntax: ";" or END expected
SYNSEMMODI, Syntax: ";", "::", "^", or "[" expected
SYNSEMRB, Syntax: ";" or "]" expected
SYNSEOTEN, Syntax: ";", OTHERWISE or END expected
SYNTHEN, Syntax: THEN expected
SYNTODOWN, Syntax: TO or DOWNTO expected
SYNTYPCNF, Syntax: type identifier or conformant schema expected
SYNTYPID, Syntax: type identifier expected

Error. The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

SYNTYPPACK, Only ARRAY, FILE, RECORD or SET types can be PACKED

Warning. You cannot pack any type other than the structured types listed in the message.

SYNTYPSPEC, Syntax: type specification expected
SYNUNEXDECL, Syntax: declaration encountered in executable section
SYNUNTIL, Syntax: UNTIL expected
SYNXTRASEMI, Syntax: "; ELSE" is not valid Pascal, ELSE matched with IF on line "line number"

Error. The compiler either failed to find an important lexical or syntactical element where one was expected, or it detected an error in such an element that does exist in your program.

TAGNOTORD, Tag type must be an ordinal type

Error. The type of a variant record's tag field must be one of the ordinal types.

TOOIDXEXPR, Too many index expressions; type has only "number of dimensions" dimensions

Error. A call to the UPPER or LOWER function specified an index value that exceeds the number of dimensions in the dynamic array.

TYPFILSIZ, Type contains one or more FILE components, size attribute is illegal

Error. The allocation size of a FILE type cannot be controlled by a size attribute; therefore, you cannot use a size attribute on any type that has a file component.

TYPHASFILE, Type contains one or more FILE components

Error. Many operations are illegal on objects of type FILE and objects of structured types with file components; for example, you cannot initialize them, use them as value parameters, or read them with input procedures.

TYPHASNOVRNT, Type contains no variant part

Error. The formats of the NEW, DISPOSE, and SIZE routines that allow case labels to be specified can be used only when their parameters have variants.

TYPPTRFIL, Type must be pointer or FILE

Error. You cannot use the syntax "Variable^" to refer to an object whose type is not pointer or FILE.

TYPSTDESCR, %STDESCR not allowed for this type

Error. The %STDESCR mechanism specifier is allowed only on objects of type CHAR, PACKED ARRAY [1..n] OF CHAR, VARYING OF CHAR, and arrays of these types.

TYPVARYCHR, Component type of VARYING must be CHAR

Error.

UNALIGNED, "variable name" is UNALIGNED

Error or Warning. You cannot use the data items listed in a call to the ADDRESS function, nor can you pass them as writeable VAR, %REF, %DESCR, or %STDESCR parameters. This message is at warning level if the variable or component has the UNALIGNED attribute, and at error level if the variable or component is actually unaligned.

UNBPNTRET, "routine name" is not UNBOUND—only 32-bit address of entry point returned

Warning. The IADDRESS function returns an address as an integer. If you pass it the name of a routine, IADDRESS returns only the first 32 bits of the routine's bound procedure value.

UNCERTAIN, "Variable name" has not been initialized

Information.

UNDECLFRML, Undeclared formal parameter "symbol name"

Error. A formal parameter name listed in a nonpositional call to a routine does not match any of the formal parameters declared in the routine heading.

UNDECLID, Undeclared identifier "symbol name"

Error. In Pascal, an identifier must be declared before it is used. There are no default or implied declarations.

UNINIT, "Variable name" may not have been initialized

Warning.

UNREAD, Variable, "variable name" is assigned into, but never read

Information.

UNSCNFVRY, UNSAFE attribute not allowed on conformant VARYING schema

Error.

UNSEXCRNG, UNSIGNED constant exceeds range

Error. The largest value allowed for an UNSIGNED integer is 4,294,967,295.

UNUSED, Variable, "variable name" is never referenced

Information.

UNWRITTEN, Variable "variable name" is read, but never assigned into

Warning.

UPLEVELACC, Unbound "routine name" precludes uplevel access to "variable name"

Error. A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

UPLEVELGOTO, Unbound "routine name" precludes uplevel GOTO to "label name"

Error. A routine that was declared with the UNBOUND attribute cannot refer to automatic variables, routines, or labels declared in outer blocks.

USEDBFDECL, "symbol name" was used before being declared

Warning.

V1DYNARR, Decommitted Version 1 dynamic array type

Warning. The type syntax used to define a dynamic array parameter has been decommitted for the current version of VAX PASCAL. You should edit your program to make the type definition conform to the current version conformant array syntax.

V1DYNARRASN, Decommitted Version 1 dynamic array assignment

Warning. In Version 1, dynamic arrays used in assignments could not be checked for compatibility until run time. This warning indicates that your program depends on an obsolete feature, which you should consider changing to reflect the current version syntax for conformant array parameters.

V1MISSPARM, Decommitted missing parameter syntax: correct by adding "number of commas" comma(s)

Warning. An OPEN procedure called with the decommitted Version 1 syntax fails to mark omitted parameters with commas. Your program depends on this obsolete feature, and you should insert the correct number of commas as listed in the message.

V1PARMSYN, Use of unsupported V1 omitted parameter syntax with a Version 3 feature(s)

Error. In a parameter list for the OPEN procedure, you cannot use both the Version 1 syntax for OPEN and the parameters that are new to subsequent versions of VAX PASCAL.

V1RADIX, Decommitted Version 1 radix output specification

Information. In Version 1, octal and hexadecimal values could be written by placing the keywords OCT or HEX after a field width expression. Your program uses this obsolete feature; you should consider changing it to use the current versions OCT or HEX predeclared functions.

VALOUTBND, Value to be assigned is out of bounds

Error. A value specified in an array or record constructor exceeds the subrange defined as the type of the corresponding component.

VALUEINIT, VALUE variables must be initialized

Error. Variables with both the VALUE and GLOBAL attributes must be given an initial value in either the VAR section or in the VALUE section.

VALUETOOBIG, VALUE attribute not allowed on objects larger than 32 bits

Error. Variables with the VALUE attribute cannot be larger than 32 bits because they are expressed to the linker as global symbol references.

VALUETYP, VALUE allowed only on ordinal or real types

Error.

VALUEVISIB, GLOBAL or EXTERNAL visibility is required with the VALUE attribute

Error. Variables with the VALUE attribute must be given either external or global visibility. (If the variable is given global visibility, then it must also be given an initial value.)

VARCOMFRML, Variable is not compatible with formal parameter "formal parameter name"

Error. A variable being passed as an actual parameter is not compatible with the corresponding formal parameter indicated. Variable parameters must be structurally compatible. The reason for the incompatibility is provided in an informational message that the compiler prints along with this error message.

VARNOTEXT, Variable must be of type TEXT

Error. The EOLN function requires that its parameter be a file of type TEXT.

VARPRMRTN, Formal VAR parameter may not be a routine

Error. The reserved word VAR cannot precede the word PROCEDURE or FUNCTION in a formal parameter declaration.

VARPTRTYP, Variable must be of a pointer type

Error. The NEW and DISPOSE procedures operate only on pointer variables.

VARYFLDS, LENGTH and BODY are the only fields in a VARYING type

Error. You cannot use the syntax "Variable.Identifier" to specify any fields of a VARYING OF CHAR variable other than LENGTH and BODY.

VISAUTOCON, Visibility / AUTOMATIC allocation conflict

Error. The GLOBAL, EXTERNAL, WEAK_GLOBAL, and WEAK_EXTERNAL attributes require static allocation and therefore conflict with the AUTOMATIC attribute.

VISGLOBEXT, Visibilities are not GLOBAL/EXTERNAL or EXTERNAL/EXTERNAL

Information. In repeated declarations of a variable or routine, only one declaration at most can be global; all others must be external. This message can appear as additional information for other error messages.

WDTHONREAL, Second field width is allowed only when value is of a real type

Error. The fraction value in a field-width specification is allowed only for real-number values.

WRITEONLY, "variable name" is WRITEONLY

Warning. You cannot use a write-only variable in any context that requires the variable to be evaluated. For example, a write-only variable cannot be used as the control variable of a FOR statement.

XTRAERRORS, Additional diagnostics occurred on this line

Information. The number of errors occurring on this line exceeds the implementation's limit for outputting errors. You should correct the errors given and recompile your program.

A.2 Run-Time Diagnostics

When an error occurs at run time, the VAX PASCAL run-time system issues an error message and aborts program execution. The syntax of the error message is as follows:

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

code
An abbreviation of the message text. Messages are alphabetized in this appendix by this code.

text
The explanation of the error.

Some conditions, particularly I/O errors, may cause several messages to be generated. The first message is a diagnostic that specifies the file that was being accessed (if any) when the error occurred and the nature of the error. Next, an RMS error message may be generated. In most cases, you should be able to understand the error by looking up the first message in the following list. If not, see the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for an explanation of the RMS error message.

Quotation marks in message text enclose items for which the compiler will replace with the name of a data object when it generates the message.

ACCMETINC, ACCESS_METHOD specified is incompatible with this file

> **Explanation.** The value of the ACCESS_METHOD parameter for a call to the OPEN procedure is not compatible with the file's organization or record type. You can use DIRECT access only with files that have relative organization or sequential organization and fixed-length records. You can use KEYED access only with indexed files.
>
> **User Action.** Make sure that you are accessing the correct file. See Chapter 6 of this manual to determine which access method you should use.

AMBVALENU, "string" is an ambiguous value for enumerated type "type"

> **Explanation.** While a value of an enumerated type was being read from a text file, not enough characters of the identifier were found to specify an unambiguous value.
>
> **User Action.** Specify enough characters of the identifier so that it is not ambiguous.

ARRINDVAL, array index value is out of range

> **Explanation.** You enabled bounds checking for a compilation unit and attempted to specify an index that is outside the array's index bounds.
>
> **User Action.** Correct the program or data so that all references to array indexes are within the declared bounds.

ARRNOTCOM, conformant array is not compatible

Explanation. You attempted to assign one dynamic array to another that did not have the same index bounds. This error occurs only when the arrays use the decommitted Version 1 syntax for dynamic array parameters.

User Action. Correct the program so that the two dynamic arrays have the same index bounds. You could also change the arrays to conform to the current syntax for conformant arrays; most incompatibilities could then be detected at compile time rather than at run time. See the *VAX PASCAL Reference Manual* for more information on current conformant arrays.

ARRNOTSTR, conformant array is not a string

Explanation. In a string operation, you used a conformant PACKED ARRAY OF CHAR value whose index had a lower bound not equal to 1 or an upper bound greater than 65535.

User Action. Correct the array's index so that the array is a character string.

BUGCHECK, internal consistency failure "nnn" in Pascal Run-Time Library

Explanation. The VAX PASCAL Run-Time Library has detected an internal error or inconsistency. This problem may be caused by an out-of-bounds array reference or a similar error in your program.

User Action. Rerun your program with all CHECK options enabled. If you are unable to find an error in your program, please submit a Software Performance Report (SPR) to DIGITAL, including a machine-readable copy of your program, data, and a sample execution illustrating the problem.

BADBITARG, Nbits argument to DEC or UDEC must be between 1 and 32

Explanation. The value being passed to DEC or UDEC exceeds the boundaries of a longword (32 bits).

User Action. If you are calling PAS$DEC or PAS$UDEC directly, examine your program for bad data. If PAS$DEC or PAS$UDEC have been called through predeclared routines in your program, then submit a Software Performance Report (SPR).

CANCNTERR, handler cannot continue from a nonfile error

Explanation. A user condition handler attempted to return SS$_CONTINUE for an error not involving file input/output. To recover from such an error you must use either an uplevel GOTO statement or the SYS$UNWIND system service.

User Action. Modify the user handler to use one of the allowed recovery actions for nonfile errors, or to resignal the error if no recovery action is possible.

CASSELVAL, CASE selector value is out of range

Explanation. The value of the case selector in a CASE statement does not equal any of the specified case labels, and the statement has no OTHERWISE clause.

User Action. Either add an OTHERWISE clause to the CASE statement or change the value of the case selector so that it equals one of the case labels. See the *VAX PASCAL Reference Manual* for more information.

CONCATLEN, string concatenation has more than 65535 characters

Explanation. The result of a string concatenation operation would result in a string longer than 65,535 characters, which is the maximum length of a string.

User Action. Correct the program so that all concatenations result in strings no longer than 65,535 characters.

CURCOMUND, current component is undefined for DELETE or UPDATE

Explanation. You attempted a DELETE or UPDATE procedure when no current component was defined. A current component is defined by a successful GET, FIND, FINDK, RESET, or RESETK that locks the component. Files opened with HISTORY:=READONLY never lock components.

User Action. Correct the program so that a current component is defined before executing DELETE or UPDATE.

DELNOTALL, DELETE is not allowed for a sequential organization file

Explanation. You attempted a DELETE procedure for a file with sequential organization, which is not allowed. DELETE is valid only on files with relative or indexed organization.

User Action. Make sure that the program is referencing the correct file. See Chapter 6 to determine what file characteristics are appropriate for your application.

ERRDURCLO, error during CLOSE

Explanation. RMS reported an unexpected error during execution of the CLOSE procedure. The RMS error message is also displayed. This message may also be issued with error severity when files are implicitly closed during a procedure or image exit.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDEL, error during DELETE

Explanation. RMS reported an unexpected error during execution of a DELETE procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURDIS, error during DISPOSE

Explanation. An error occurred during execution of a DISPOSE procedure. An additional message that further describes the error may also be displayed.

User Action. Make sure that the heap storage being freed was allocated by a successful call to the NEW procedure, and that it has not been already freed. If an additional message is shown, see the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of that message.

ERRDUREXT, error during EXTEND

Explanation. RMS reported an unexpected error during execution of an EXTEND procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURFIN, error during FIND or FINDK

Explanation. RMS reported an unexpected error during execution of a FIND or FINDK procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURGET, error during GET

Explanation. RMS reported an unexpected error during execution of the GET procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for the description of the RMS error.

ERRDURMAR, error during MARK

Explanation. An error occurred during execution of the PAS$MARK2 procedure. An additional message is displayed that further describes the error.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURNEW, error during NEW

Explanation. An error occurred during execution of the NEW procedure. An additional message is displayed that further describes the error.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDUROPE, error during OPEN

Explanation. An unexpected error occurred during execution of the OPEN procedure, or during an implicit open caused by a RESET or REWRITE procedure. An additional message is displayed that further describes the error.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional message.

ERRDURPRO, error during prompting

Explanation. RMS reported an unexpected error during output of partial lines to a terminal. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURPUT, error during PUT

Explanation. RMS reported an unexpected error during execution of the PUT procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS message.

ERRDURREL, error during RELEASE

Explanation. An unexpected error occurred during execution of the PAS$RELEASE2 procedure. An additional message may be displayed that further describes the error.

User Action. Make sure that the marker argument was returned from a successful call to PAS$MARK2 and that the storage has not been already freed. If an additional message is displayed, see the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of that message.

ERRDURRES, error during RESET or RESETK

Explanation. RMS reported an unexpected error during execution of the RESET or RESETK procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURREW, error during REWRITE

Explanation. RMS reported an unexpected error during execution of the REWRITE procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURTRU, error during TRUNCATE

Explanation. RMS reported an unexpected error during execution of the TRUNCATE procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUNL, error during UNLOCK

Explanation. RMS reported an unexpected error during execution of the UNLOCK procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURUPD, error during UPDATE

Explanation. RMS reported an unexpected error during execution of the UPDATE procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

ERRDURWRI, error during WRITELN

Explanation. RMS reported an unexpected error during execution of the WRITELN procedure. The RMS error message is also displayed.

User Action. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the RMS error.

EXTNOTALL, EXTEND is not allowed for a shared file

Explanation. Your program attempted an EXTEND procedure for a file for which the program did not have exclusive access. EXTEND requires that no other users be allowed to access the file. Note that this message may also be issued if you do not have permission to extend to the file.

User Action. Correct the program so that the file is opened with SHARING:=NONE, which is the default, before performing an EXTEND procedure.

FAIGETLOC, failed to GET locked component

Explanation. Your program attempted to access a component of a file that was locked by another user. You can usually expect this condition to occur when more than one user is accessing the same relative or indexed file.

User Action. Determine whether this condition should be allowed to occur. If so, modify your program so that it detects the condition and retries the operation later. See Chapter 6 for more information.

FILALRACT, file "file name" is already active

Explanation. Your program attempted a file operation on a file for which another operation was still in progress. This error can occur if a file is used in AST or condition-handling routines.

User Action. Modify your program so that it does not try to use files that may currently be in use.

FILALRCLO, file is already closed

Explanation. Your program attempted to close a file that was already closed.

User Action. Modify your program so that it does not try to close files that are not open.

FILALROPE, file is already open

Explanation. Your program attempted to open a file that was already open.

User Action. Modify your program so that it does not try to open files that are already open.

FILNAMREQ, FILE_NAME required for this HISTORY or DISPOSITION

Explanation. Your program attempted to open a nonexternal file without specifying a file name parameter to the OPEN procedure, but the HISTORY or DISPOSITION parameter specified requires a file name. If HISTORY is OLD or READONLY, or if DISPOSITION is PRINT, PRINT_DELETE, SUBMIT, or SUBMIT_DELETE, you must also specify a file name parameter.

User Action. Add a file name parameter to the OPEN procedure call, specifying an appropriate file name.

FILNOTDIR, file is not opened for direct access

Explanation. Your program attempted to execute a DELETE, FIND, LOCATE, or UPDATE procedure on a file that was not opened for direct access.

User Action. Modify the program to specify the ACCESS_METHOD:=DIRECT parameter to the OPEN procedure when opening the file. See Chapter 6 to determine if direct access is appropriate for your application.

FILNOTFOU, file not found

Explanation. Your program attempted to open a file that does not exist. An additional RMS message is displayed that further describes the problem.

User Action. Make sure that you are specifying the correct file. See the *VAX/VMS System Messages and Recovery Procedures Reference Manual* for a description of the additional RMS message.

FILNOTGEN, file is not in Generation mode

Explanation. Your program attempted a file operation that required the file to be in generation mode (ready for writing).

User Action. Modify the program to use a REWRITE, TRUNCATE, or LOCATE procedure to place the file in generation mode as appropriate. See Chapter 6 for more information.

FILNOTINS, file is not in Inspection mode

Explanation. Your program attempted a file operation that required the file to be in inspection mode (ready for reading).

User Action. Modify the program to use a RESET, RESETK, FIND, or FINDK procedure to place the file in inspection mode as appropriate. See Chapter 6 for more information.

FILNOTKEY, file is not opened for keyed access

Explanation. Your program attempted to execute a FINDK, RESETK, DELETE, or UPDATE procedure on a file that was not opened for keyed access.

User Action. Modify the program to specify the ACCESS_METHOD:=KEYED parameter to the OPEN procedure when opening the file. See Chapter 6 to make sure that keyed access is appropriate to your application.

FILNOTOPE, file is not open

Explanation. Your program attempted to execute a file manipulation procedure on a file that was not open.

User Action. Correct the program to open the file using a RESET, REWRITE, or OPEN procedure as appropriate. See Chapter 6 for more information.

FILNOTSEQ, file is not sequential organization

> **Explanation.** Your program attempted to execute the TRUNCATE procedure on a file that does not have sequential organization. TRUNCATE is valid only on sequential files.
>
> **User Action.** Make sure that your program is accessing the correct file. Correct the program so that all TRUNCATE operations are performed on sequential files.

FILNOTTEX, file is not a textfile

> **Explanation.** Your program performed a file operation that required a file of type TEXT on a nontext file. Note that the type FILE OF CHAR is not equivalent to TEXT unless you have compiled the program with the /OLD_VERSION qualifier.
>
> **User Action.** Make sure that your program is accessing the correct file. Correct the program so that a text file is always used when required.

GENNOTALL, Generation mode is not allowed for a READONLY file

> **Explanation.** Your program attempted to place a file declared with the READONLY attribute into generation mode, which is not allowed. Note that the READONLY file attribute is not equivalent to the HISTORY:=READONLY parameter to the OPEN procedure.
>
> **User Action.** Correct the program so that the file either does not have the READONLY attribute or is not placed into generation mode.

GETAFTEOF, GET attempted after end-of-file

> **Explanation.** Your program attempted a GET operation on a file while EOF(f) was TRUE. This situation occurs when a previous GET operation (possibly implicitly performed by a RESET, RESETK, or READ procedure) reads to the end of the file and causes the EOF(f) function to return TRUE. If another GET is then performed, this error is given.
>
> **User Action.** Correct the program so that it either tests whether EOF(f) is TRUE, before attempting a GET operation, or repositions the file before the end-of-file marker.

GOTOFAILED, non-local GOTO failed

Explanation. An error occurred while a nonlocal GOTO statement was being executed. This error might occur because of an error in the user program, such as an out-of-bounds array reference.

User Action. Rerun your program, enabling all CHECK options. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

HALT, HALT procedure called

Explanation. The program terminated its execution by executing the HALT procedure. This message is solely informational.

User Action. None.

INSNOTALL, Inspection mode is not allowed for a WRITEONLY file

Explanation. Your program attempted to place a file declared with the WRITEONLY attribute into inspection mode, which is not allowed.

User Action. Correct the program so that the file variable either does not have the WRITEONLY attribute or is not placed into inspection mode.

INSVIRMEM, insufficient virtual memory

Explanation. The VAX/VMS Run-Time Library was unable to allocate enough heap storage to open the file.

User Action. Examine your program to see whether it is making excessive use of heap storage, which might be allocated using the NEW procedure or the Run-Time Library procedure LIB$GET_VM. Modify your program to free any heap storage it does not need.

INVARGPAS, invalid argument to Pascal Run-Time Library

Explanation. An invalid argument or inconsistent data structure was passed to the VAX/VMS Run-Time Library by the compiled code, or a system service returned an unrecognized value to the Run-Time Library.

User Action. Rerun your program with all CHECK options enabled. Make sure that the version of the current operating system is compatible with the version of the compiler. If you cannot locate an error in your program and the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

INVFILSYN, invalid file name syntax

Explanation. Your program attempted to open a file with an invalid file name. The file name used can be derived from the file variable name, the value of the file name parameter to the OPEN procedure, or the logical name translations (if any) of the file variable name and portions of the file name parameter and your default device and directory. The displayed text may include the erroneous file name. This error can also occur if the value of the file name parameter is longer than 255 characters. Additional RMS messages may be displayed that further describe the error.

User Action. Use the information provided in the displayed messages to determine which component of the file name is invalid. Verify that any logical names used are defined correctly. See the *VAX PASCAL Reference Manual* for information on file names.

INVFILVAR, invalid file variable at location "nnn"

Explanation. The file variable passed to a VAX/VMS Run-Time Library procedure was invalid or corrupted. This problem might be caused by an error in the user program, such as an out-of-bounds array access. It can also occur if a file variable is passed from a routine compiled with a version of VAX PASCAL earlier than Version 2 to a routine compiled with a later version of the compiler, or if the new key options are used on VAX/VMS systems earlier than V4.6.

User Action. Rerun your program with all CHECK options enabled, and recompile all modules using the same compiler.

If the problem persists, please submit a Software Performance Report (SPR) to DIGITAL, and include a machine-readable copy of your program, data, and results of a sample execution illustrating the problem.

INVKEYDEF, invalid key definition

Explanation. Your program attempted to open a file of type RECORD whose component type contained a field with an invalid KEY attribute. One of the following errors occurred:

- A new file was being created and the key numbers were not dense.
- A key field was defined at an offset of more than 65,535 bytes from the beginning of the record.

User Action. If a new file is being created, make sure that the key fields are numbered consecutively, starting with 0 for the required primary key. If you are opening an existing file, you must explicitly specify HISTORY:=OLD or HISTORY:=READONLY as a parameter to the OPEN procedure. Make sure that the length of the record is within the maximum permitted for the file organization being used. See Chapter 6 for more information.

INVRECLEN, invalid record length of "nnn"

Explanation. A file was being opened, and one of the following errors occurred:

- The length of the file components was greater than 65,535 bytes.
- The value of the RECORD_LENGTH parameter to the OPEN procedure was greater than 65,535.

User Action. Correct the program so that the record length used is within the permitted limits for the type of file being used. See Chapter 6 for more information.

INVSYNBIN, "string" is invalid syntax for a binary value

Explanation. While a READ or READV procedure was reading a binary value from a text file, the characters read did not conform to the syntax for a binary value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNHEX, "string" is invalid syntax for a hexadecimal value

Explanation. While a READ or READV procedure was reading a hexadecimal value from a text file, the characters read did not conform to the syntax for an hexadecimal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNENU, "string" is invalid syntax for an enumerated value

Explanation. While a READ or READV procedure was reading an identifier of an enumerated type from a text file, the characters read did not conform to the syntax for an enumerated value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNINT, "string" is invalid syntax for an integer value

Explanation. While a READ or READV procedure was reading a value for an integer identifier from a text file, the characters read did not conform to the syntax for an integer value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNOCT, "string" is invalid syntax for an octal value

Explanation. While a READ or READV procedure was reading an octal value from a text file, the characters read did not conform to the syntax for an octal value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNREA, "string" is invalid syntax for a real value

Explanation. While a READ or READV procedure was reading a value for a real identifier from a text file, the characters read did not conform to the syntax for a real value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

INVSYNUNS, "string" is invalid syntax for an unsigned value

Explanation. While a READ or READV procedure was reading a value for an unsigned identifier from a text file, the characters read did not conform to the syntax for an unsigned value. The displayed message includes the text actually read and the record number in which this text occurred.

User Action. Correct the program or the input data so that the correct syntax is used. See the *VAX PASCAL Reference Manual* for more information.

KEYCHANOT, key field change is not allowed

Explanation. Your program attempted an UPDATE procedure for a record of an indexed file that would have changed the value of a key field, and this situation was disallowed when the file was created.

User Action. If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly change a key field, or recreate the file specifying that the key field is permitted to change. See Chapter 6 for more information.

KEYDEFINC, KEY "nnn" definition is inconsistent with this file

Explanation. An indexed file of type RECORD was opened, and the component type contained fields whose KEY attributes did not match those of the existing file. The number of the key in error is displayed in the message.

User Action. Correct the RECORD definition so that it describes the correct KEY fields, or recreate the file so that it matches the declared keys. See Chapter 6 for more information.

KEYDUPNOT, key field duplication is not allowed

Explanation. Your program attempted an UPDATE or PUT procedure for a record of an indexed file that would have duplicated a key field value of an existing record, and this situation was disallowed when the file was created.

User Action. If the program needs to detect this situation when it occurs, specify the ERROR:=CONTINUE parameter for the PUT or UPDATE procedure, and use the STATUS function to determine which error, if any, occurred. If necessary, modify the program so that it does not improperly duplicate a key field, or recreate the file specifying that the key field is permitted to be duplicated. See Chapter 6 for more information.

KEYNOTDEF, KEY "nnn" is not defined for this file

Explanation. Your program attempted a FINDK or RESETK procedure on an indexed file, and the key number specified does not exist in the file.

User Action. Correct the program so that the correct key numbers are used when accessing the file.

KEYVALINC, key value is incompatible with the file's key "nnn"

Explanation. The key value specified for the FINDK procedure was incompatible in type or size with the key field of the file.

User Action. Make sure that the correct key value is being specified for FINDK. Correct the program so that the type of the key value is compatible with the key of the file. See the *VAX PASCAL Reference Manual* for more information.

LINTOOLON, line is too long, exceeded record length by "nnn" character(s)

Explanation. Your program attempted a WRITE, PUT, WRITEV, or other output procedure on a text file that would have placed more characters in the current line than the record length of the file would allow. The number of characters that did not fit is displayed in the message.

User Action. Correct the program so that it does not place too many characters in the current line. If appropriate, use the WRITELN procedure, or specify an increased record length parameter when opening the file with the OPEN procedure.

LINVALEXC, LINELIMIT value exceeded

Explanation. The number of lines written to the file exceeded the maximum specified as the line limit. The line limit value is determined by the translation of the logical name PAS$LINELIMIT, if any, or the value specified in a call to the LINELIMIT procedure for the file.

User Action. As appropriate, correct the program so that it does not write as many lines, or increase the line limit for the file. Note that if a line limit is specified for a nontext file, each PUT procedure called for the file is considered to be one line. See the *VAX PASCAL Reference Manual* for more information.

MODNEGNUM, MOD of a negative modulus has no mathematical definition

Explanation. In the MOD operation A MOD B, the operand B must have a positive integer value.

User Action. Correct the program so that the operand B has a positive integer value.

NEGDIGARG, negative Digits argument to BIN, HEX or OCT is not allowed

Explanation. Your program attempted to specify a negative value for the Digits argument in a call to the BIN, HEX, or OCT procedure, which is not permitted.

User Action. Correct the program so that only nonnegative Digits arguments are used for calls to BIN, HEX, and OCT.

NEGWIDDIG, negative Width or Digits specification is not allowed

Explanation. A WRITE or WRITEV procedure on a text file contained a field width specification that included a negative width or digits value, which is not permitted.

User Action. Correct the program so that only nonnegative Width and Digits parameters are used.

NOTVALTYP, "string" is not a value of type "type"

Explanation. Your program attempted a READ or READV procedure on a text file, but the value read could not be expressed in the specified type. For example, this error results if a real value read is outside the range of the identifier's type, or if an enumerated value is read that does not match any of the valid constant identifiers in its type.

User Action. Correct the program or the input data so that the values read are compatible with the types of the identifiers receiving the data.

ORDVALOUT, ordinal value is out of range

Explanation. A value of an ordinal type is outside the range of values specified by the type. For example, this error results if you try to use the SUCC function on the last value in the type or the PRED function on the first value.

User Action. Correct the program so that all ordinal values are within the range of values specified by the ordinal type.

ORGSPEINC, ORGANIZATION specified is inconsistent with this file

Explanation. The value of the ORGANIZATION parameter for the OPEN procedure that opened an existing file was inconsistent with the actual organization of the file.

User Action. Correct the program so that the correct organization is specified. See Chapter 6 for more information.

PADLENERR, PAD length error

Explanation. The length of the character string to be padded by the PAD function is greater than the length specified as the finished size, or the finished size specified is greater than 65535.

User Action. Correct the call to PAD so that the finished size specified describes a character string of the correct length. See the *VAX PASCAL Reference Manual* for the rules governing the PAD function.

PTRREFNIL, pointer reference to NIL

Explanation. Your program attempted to evaluate a pointer value while its value was NIL.

User Action. Make sure that the pointer has a value before you try to evaluate it. See the *VAX PASCAL Reference Manual* for more information on pointer values.

RECLENINC, RECORD_LENGTH specified is inconsistent with this file

Explanation. The record length obtained from the file component's length or from the value of the record length parameter specified for the OPEN procedure was inconsistent with the actual record length of an existing file.

User Action. Correct the program so that the record length specified, if any, is consistent with the file. See Chapter 6 for more information.

RECTYPINC, RECORD_TYPE specified is inconsistent with this file

Explanation. The value of the RECORD_LENGTH parameter specified for the OPEN procedure was inconsistent with the actual record type of an existing file.

User Action. Correct the program so that the record type specified, if any, is consistent with the file. See Chapter 6 for more information.

RESNOTALL, RESET is not allowed on an unopened internal file

Explanation. Your program attempted a RESET procedure for a nonexternal file that was not open. This operation is not permitted because RESET must operate on an existing file, and there is no information associated with a nonexternal file that allows RESET to open it.

User Action. Correct the program so that nonexternal files are opened before using RESET. Either OPEN or REWRITE may be used to open a nonexternal file. See the *VAX PASCAL Reference Manual* for more information.

REWNOTALL, REWRITE is not allowed for a shared file

Explanation. Your program attempted a REWRITE procedure for a file for which the program did not have exclusive access. REWRITE requires that no other users be allowed to access the file while the file's data is deleted. Note that this message may also be issued if you do not have permission to write to the file.

User Action. Correct the program so that the file is opened with SHARING := NONE, which is the default, before performing a REWRITE procedure.

SETASGVAL, set assignment value has element out of range

Explanation. Your program attempted to assign to a set variable a value that is outside the range specified by the variable's component type.

User Action. Correct the assignment statement so that the value being assigned falls within the component type of the set variable. See the *VAX PASCAL Reference Manual* for more information on sets.

SETCONVAL, set constructor value out of range

Explanation. Your program attempted to include in a set constructor a value that is outside the range specified by the set's component type, or a value that is greater than 255 or less than 0.

User Action. Correct the constructor so that it includes only those values within the range of the set's component type. See the *VAX PASCAL Reference Manual* for more information on sets.

STRASGLEN, string assignment length error

Explanation. Your program attempted to assign to a string variable a character string that is longer than the declared maximum length of the variable (if the variable's type is VARYING) or that is not of the same length as the variable (if the variable's type is PACKED ARRAY OF CHAR).

User Action. Correct the program so that the string is of a correct length for the variable to which it is being assigned.

STRCOMLEN, string comparison length error

Explanation. Your program attempted to compare two character strings that do not have the same current length.

User Action. Correct the program so that the two strings have the same length at the time of the comparison.

SUBASGVAL, subrange assignment value out of range

Explanation. Your program attempted to assign to a subrange variable a value that is not contained in the subrange type.

User Action. Correct the program so that all values assigned to a subrange variable fall within the variable's type.

SUBSTRSEL, SUBSTR selection error

Explanation. A SUBSTR function attempted to extract a substring that was not entirely contained in the original string.

User Action. Correct the call to SUBSTR so that it specifies a substring that can be extracted from the original string. See the *VAX PASCAL Reference Manual* for complete information on the SUBSTR function.

TEXREQSEQ, textfiles require sequential organization and access

Explanation. Your program attempted to open a file of type TEXT that either did not have sequential organization, or had an ACCESS_METHOD other than SEQUENTIAL (the default) when opened by the OPEN procedure.

User Action. Make sure that the program refers to the correct file. Correct the program so that only sequential organization and access are used for text files.

TRUNOTALL, TRUNCATE is not allowed for a shared file

Explanation. Your program attempted to call the TRUNCATE procedure for a file that was opened for shared access. You cannot truncate files that might be shared by other users. This message may also be issued if you do not have permission to write to the file.

User Action. Correct the program so that it does not try to truncate shared files. If the file is opened with the OPEN procedure, do not specify a value other than NONE (the default) for the SHARING parameter.

UPDNOTALL, UPDATE not allowed for a sequential organization file

Explanation. Your program attempted to call the UPDATE procedure for a sequential file. UPDATE is valid only on relative and indexed files.

User Action. Correct the program so that it does not try to use UPDATE for sequential files, or recreate the file with relative or indexed organization. If you are using direct access on a sequential file, individual records can be updated with the LOCATE and PUT procedures. See Chapter 6 to determine whether a different file organization may be appropriate for your application.

VARINDVAL, VARYING index value exceeds current length

Explanation. The index value specified for a VARYING OF CHAR string is greater than the string's current length.

User Action. Correct the index value so that it specifies a legal character in the string.

WRIINVENU, WRITE of an invalid enumerated value

Explanation. Your program attempted to write an enumerated value using a WRITE or WRITEV procedure, but the internal representation of that value was outside the possible range for the enumerated type.

User Action. Verify that your program is not improperly using PRED, SUCC, or type casting to assign an invalid value to a variable of enumerated type.

Appendix B

Errors Returned by STATUS and STATUSV Functions

This appendix lists the error conditions detected by the STATUS and STATUSV functions, their symbolic names, and the corresponding values. The symbolic names and their values are defined in the file SYS$LIBRARY:PASSTATUS.PAS, which you can include with a %INCLUDE directive in a CONST section of your program. To test for a specific condition, you compare the STATUS or STATUSV return values against the value of a symbolic name.

Note that the symbolic names correspond to some of the run-time errors listed in Appendix A; however, not all run-time errors can be detected by STATUS.

There is a one-to-one correspondence between the symbolic constants returned by STATUS or STATUSV documented in PASSTATUS and the VAX condition code values in SYS$LIBRARY:PASDEF.PAS. The following routine shows how to map the return value of STATUS to its corresponding condition code located in PASDEF.PAS.

```
FUNCTION CONVERT_STATUS_TO_CONDITION(STAT:INTEGER):INTEGER;
   BEGIN
   CONVERT_STATUS_TO_CONDITION := %x218644 + STAT * 8;
   END;
```

Table B–1 lists the symbolic names and the values returned by the STATUS and STATUSV functions and explains the error condition that corresponds to each value.

Table B–1: STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ACCMETINC	5	Specified access method is not compatible with this file
PAS$K_AMBVALENU	30	"String" is an ambiguous value for the enumerated type "type"
PAS$K_CURCOMUND	73	DELETE or UPDATE was attempted while the current component was undefined
PAS$K_DELNOTALL	100	DELETE is not allowed for a file with sequential organization
PAS$K_EOF	–1	File is at end-of-file
PAS$K_ERRDURCLO	16	Error occurred while the file was being closed
PAS$K_ERRDURDEL	101	Error occurred during execution of DELETE
PAS$K_ERRDUREXT	127	Error occurred during execution of EXTEND
PAS$K_ERRDURFIN	102	Error occurred during execution of FIND or FINDK
PAS$K_ERRDURGET	103	Error occurred during execution of GET
PAS$K_ERRDUROPE	2	Error occurred during execution of OPEN
PAS$K_ERRDURPRO	36	Error occurred during prompting
PAS$K_ERRDURPUT	104	Error occurred during execution of PUT
PAS$K_ERRDURRES	105	Error occurred during execution of RESET or RESETK
PAS$K_ERRDURREW	106	Error occurred during execution of REWRITE
PAS$K_ERRDURTRU	107	Error occurred during execution of TRUNCATE
PAS$K_ERRDURUNL	108	Error occurred during execution of UNLOCK
PAS$K_ERRDURUPD	109	Error occurred during execution of UPDATE

Table B-1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_ERRDURWRI	50	Error occurred during execution of WRITELN
PAS$K_EXTNOTALL	128	EXTEND is not allowed for a shared file
PAS$K_FAIGETLOC	74	GET failed to retrieve a locked component
PAS$K_FILALRCLO	15	File is already closed
PAS$K_FILALROPE	1	File is already open
PAS$K_FILNAMREQ	14	File name must be specified in order to save, print, or submit an internal file
PAS$K_FILNOTDIR	110	File is not open for direct access
PAS$K_FILNOTFOU	3	File was not found
PAS$K_FILNOTGEN	111	File is not in generation mode
PAS$K_FILNOTINS	112	File is not in inspection mode
PAS$K_FILNOTKEY	113	File is not open for keyed access
PAS$K_FILNOTOPE	114	File is not open
PAS$K_FILNOTSEQ	115	File does not have sequential organization
PAS$K_FILNOTTEX	116	File is not a text file
PAS$K_GENNOTALL	117	Generation mode is not allowed for a read-only file
PAS$K_GETAFTEOF	118	GET attempted after end-of-file has been reached
PAS$K_INSNOTALL	119	Inspection mode is not allowed for a write-only file
PAS$K_INSVIRMEM	120	Insufficient virtual memory
PAS$K_INVARGPAS	121	Invalid argument passed to a VAX PASCAL Run-Time Library procedure
PAS$K_INVFILSYN	4	Invalid syntax for file name
PAS$K_INVKEYDEF	9	Key definition is invalid
PAS$K_INVRECLEN	12	Record length nnn is invalid
PAS$K_INVSYNBIN	37	"String" is invalid syntax for a binary value

Table B–1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_INVSYNENU	31	"String" is invalid syntax for a value of an enumerated type
PAS$K_INVSYNHEX	38	"String" is invalid syntax for a hexadecimal value
PAS$K_INVSYNINT	32	"String" is invalid syntax for an integer
PAS$K_INVSYNOCT	39	"String" is invalid syntax for an octal value
PAS$K_INVSYNREA	33	"String" is invalid syntax for a real number
PAS$K_INVSYNUNS	34	"String" is invalid syntax for an unsigned integer
PAS$K_KEYCHANOT	72	Changing the key field is not allowed
PAS$K_KEYDEFINC	10	KEY(nnn) definition is inconsistent with this file
PAS$K_KEYDUPNOT	71	Duplication of key field is not allowed
PAS$K_KEYNOTDEF	11	KEY(nnn) is not defined in this file
PAS$K_KEYVALINC	70	Key value is incompatible with file's key nnn
PAS$K_LINTOOLON	52	Line is too long; exceeds record length by nnn characters
PAS$K_LINVALEXC	122	LINELIMIT value exceeded
PAS$K_NEGWIDDIG	53	Negative value in width or digits (of a field width specification) is invalid
PAS$K_NOTVALTYP	35	"String" is not a value of type "type"
PAS$K_ORGSPEINC	8	Specified organization is inconsistent with this file
PAS$K_RECLENINC	6	Specified record length is inconsistent with this file
PAS$K_RECTYPINC	7	Specified record type is inconsistent with this file
PAS$K_RESNOTALL	124	RESET is not allowed for an internal file that has not been opened

Table B–1 (Cont.): STATUS and STATUSV Return Values

Name	Value	Meaning
PAS$K_REWNOTALL	123	REWRITE is not allowed for a file opened for sharing
PAS$K_SUCCESS	0	Last file operation completed successfully
PAS$K_TEXREQSEQ	13	Text files must have sequential organization and sequential access
PAS$K_TRUNOTALL	125	TRUNCATE is not allowed for a file opened for sharing
PAS$K_UPDNOTALL	126	UPDATE is not allowed for a file that has sequential organization
PAS$K_WRIINVENU	54	WRITE operation attempted on an invalid enumerated value

Appendix C

Entry Points to VAX PASCAL Utilities

This appendix describes the entry points to utilities in the VAX Run-Time Library that can be called as external routines by a VAX PASCAL program. These utilities allow you to access VAX PASCAL extensions that are not directly provided by the language.

C.1 PAS$FAB (f)

The PAS$FAB function returns a pointer to the RMS file access block (FAB) of file f. After this function has been called, the FAB can be used to get information about the file and to access RMS facilities not explicitly available in the VAX PASCAL language.

The component type of file f can be any type; the file must be open.

PAS$FAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_FAB  = ^FAB$TYPE;

FUNCTION PAS$FAB
   (VAR F : Unsafe_File) : Ptr_to_FAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$FAB. The type FAB$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS FAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the FAB is used, which may in turn require you to change your program. See the *VAX/VMS Run-Time Library Routines Reference Manual* for more information.

C.2 PAS$RAB (f)

The PAS$RAB function returns a pointer to the RMS record access block (RAB) of file f. After this function has been called, the RAB can be used to get information about the file and to access RMS facilities not explicitly available in the VAX PASCAL language.

The component type of file f can be any type; the file must be open.

PAS$RAB is an external function that must be explicitly declared by a declaration such as the following:

```
TYPE
   Unsafe_File = [UNSAFE] FILE OF CHAR;
   Ptr_to_RAB  = ^RAB$TYPE;

FUNCTION PAS$RAB
   (VAR F : Unsafe_File) : Ptr_to_RAB;
   EXTERN;
```

This declaration allows a file of any type to be used as an actual parameter to PAS$RAB. The type RAB$TYPE is defined in the VAX PASCAL environment file STARLET.PEN, which your program or module can inherit.

You should take care that your use of the RMS RAB does not interfere with the normal operations of the Run-Time Library. Future changes to the Run-Time Library may change the way in which the RAB is used, which may in turn require you to change your program. See the *VAX/VMS Run-Time Library Routines Reference Manual* for more information.

C.3 PAS$MARK2 (s)

The PAS$MARK2 function returns a pointer to a heap-allocated object of the size specified by s. If this pointer value is then passed to the PAS$RELEASE2 function, all objects allocated with NEW or PAS$MARK2 since the object was allocated are deallocated. PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VAX PASCAL. Their use is not recommended in a modular programming environment.

While a mark is in effect, any DISPOSE operation will not actually delete the storage, but merely mark the storage for deletion. To free the memory, you must use PAS$RELEASE2.

PAS$MARK2 is an external function that must be explicitly declared. Because the parameter to PAS$MARK2 is the size of the object (unlike the parameter to the predeclared procedure NEW), the best method for using this function is to declare a separate function name for each object you wish to mark. The following example shows how PAS$MARK2 could be declared and used as a function named Mark_Integer to allocate and mark an integer variable:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer: Ptr_to_Integer;


[EXTERNAL(PAS$MARK2)] FUNCTION Mark_Integer
   (%IMMED S : Integer := SIZE(Integer))
   : Ptr_to_Integer;
   EXTERN;
   .
   .
   .
Marked_Integer := Mark_Integer;
```

The parameter to PAS$MARK2 can be 0, in which case the function value is only a pointer to a marker, and cannot be used to store data.

C.4 PAS$RELEASE2 (p)

The PAS$RELEASE2 function deallocates all storage allocated by NEW or PAS$MARK2 since the call to PAS$MARK2 that allocated the parameter p.

PAS$MARK2 and PAS$RELEASE2 are provided only for compatibility with some other implementations of VAX PASCAL. Their use is not recommended in a modular programming environment. PAS$RELEASE2 disables AST delivery during its execution, and thus should not be used in a real-time environment.

PAS$RELEASE2 is an external function that must be explicitly declared. An example of its declaration and use is as follows:

```
TYPE
   Ptr_to_Integer = ^Integer;

VAR
   Marked_Integer : Ptr_to_Integer;

[EXTERNAL(PAS$RELEASE2)] PROCEDURE Release
   (P :[UNSAFE] Ptr_to_Integer);
   EXTERN;
   .
   .
   .
Release (Marked_Integer);
```

In this example, Marked_Integer is assumed to contain the pointer value returned by a previous call to PAS$MARK2. See Section C.3 for information about PAS$MARK2.

Appendix D

Differences Between Version 1 and Subsequent Versions

This appendix describes the differences between VAX PASCAL Version 1 and all subsequent higher versions. In this appendix, the term Version 2+ will refer to both Version 2 and Version 3. The differences between Version 1 and Version 2+ fall into three categories:

- Features that have been decommitted. The previous versions of these features are still supported in Version 2+ to allow you to run existing programs; however, it is recommended that you modify your programs to reflect the new changes.
- Features that are controlled by the /OLD_VERSION compile-time qualifier.
- Minor changes that are not likely to affect the vast majority of existing VAX PASCAL programs.

NOTE

This appendix is intended for users migrating from Version 1 to Version 3. If you are migrating from Version 2 to a Version 3, you can disregard this appendix, as there have been no incompatible changes made between Version 2 and Version 3.

If you modify a program that executed successfully under Version 1 of VAX PASCAL, you should not make changes that conflict with the Version 2+ standard. If conflicts exist and you compile the program with Version 2+, one of two problems may result:

- You may get warning messages at compile time.

- The program may compile successfully but may not run.

If you must use language features that conflict with Version 2+, you can use the /OLD_VERSION qualifier at compile time to produce the desired results. The /OLD_VERSION qualifier and the conflicts that it resolves are described in Section D.2.

D.1 Decommitted Features

The following decommitted features are described in this section:

- Syntax of dynamic array parameters
- Predeclared functions LOWER and UPPER
- Printing of hexadecimal and octal values with the WRITE procedure
- Syntax of the OPEN procedure
- Specification of compiler qualifiers in the source code

D.1.1 Dynamic Array Parameters

Some programming applications require general routines that can process arrays with different bounds. Version 1 of VAX PASCAL allows you to declare routines with dynamic array parameters. You can call the routine with arrays of different sizes, as long as their bounds are within those specified by the formal parameter.

For example, you can write a procedure that sums the components of a one-dimensional array. Each time you use the procedure, you might want to pass arrays with different bounds. Instead of declaring multiple procedures using arrays of each possible size, you could use a dynamic array parameter. The procedure will treat the formal parameter as though its bounds were those of the actual parameter.

In routines that contain dynamic array parameters, you use the predeclared functions LOWER and UPPER to return the lower and upper bounds of the actual array parameter (see Section D.1.2). An array parameter has the following form:

```
array-identifier,... : PACKED ARRAY
                          [{index-type-identifier},...]
                          OF type-identifier
```

Note that you must use a type identifier to specify the range of the indexes. You cannot use a subrange. The type identifier can be any of the predefined ordinal types (for example, INTEGER).

The components and indexes of the actual and formal dynamic array parameters must be of compatible types. The rules for dynamic array compatibility are identical to those for compatibility between other arrays, with one exception: the range of the index types of the actual array parameter must be within the range specified for the formal parameter.

The following differences exist between Version 1 and Version 2+ syntax. See the *VAX PASCAL Reference Manual* for further details on syntax.

- In Version 2+, dynamic arrays are known as conformant arrays, and the syntax that describes them is called a conformant array schema.
- The conformant array schema for Version 2+ requires that the upper and lower bounds of the conformant array parameter be declared with identifiers in the formal parameter list. You can then use these identifiers within the routine block to refer to the upper and lower bounds of the parameter.
- Version 2+ allows the type identifier of a conformant array parameter to be another conformant array schema.

D.1.2 LOWER and UPPER Functions

Version 1 of VAX PASCAL includes the predeclared functions LOWER and UPPER, which you can use to determine the upper and lower bounds of dynamic array parameters (see Section D.1.1). Because the syntax of conformant array parameters has changed (see the *VAX PASCAL Reference Manual*), these functions are no longer necessary. They are supported, however, for programs that use the old syntax. These functions have the following form:

```
LOWER (a [[, n]])
UPPER (a [[, n]])
```

The parameter a denotes an array variable; the optional parameter n is an integer constant that denotes a dimension of a. If you omit a value for the parameter n, it defaults to 1. The LOWER function returns the lower bound of the nth dimension of a; the UPPER function returns the upper bound of the nth dimension of a.

D.1.3 Printing Hexadecimal and Octal Values

Version 2+ provides the predeclared functions HEX, OCT, and BIN, which return the hexadecimal, octal, and binary equivalents of the input value (see the *VAX PASCAL Reference Manual*). You can use these functions in conjunction with the WRITE, WRITELN, and WRITEV procedures to print values in hexadecimal, octal, and binary notation.

The following formats of the WRITE procedure are used to print hexadecimal and octal values in Version 1.

```
WRITE (expression:field-width HEX,...)

WRITE (expression:field-width OCT,...)
```

expression
The value to be written. Arbitrary items (including pointers) can be written in hexadecimal or octal notation to text files.

field-width
A positive integer expression indicating the length of the print field.

For hexadecimal values, if the field width specified is less than eight characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than eight characters, and the output value is less than the field width, the field is padded with blanks on the left.

For octal values, if the field width specified is less than 11 characters, and the output value is greater than the field width, the value being printed is truncated on the left. If the field width is greater than 11 characters, and the output value is less than the field width, the field is padded with blanks on the left.

Example 1

```
WRITE (Payroll:10 HEX);
```

The value of the variable Payroll is printed in a field of 10 hexadecimal characters.

Example 2

```
WRITE (Social_Security:14 OCT);
```

The value of the variable Social_Security is printed in a field of 14 octal characters.

D.1.4 The OPEN Procedure

The OPEN procedure opens a file and allows you to specify file parameters. Version 2+ includes new parameters and additional parameter values and has changed some defaults. Table D–1 lists the file parameters available under Version 1, their possible values, and their defaults.

Table D–1: Summary of Version 1 OPEN Parameters

Parameter	Parameter Values	Default
History	OLD or NEW	NEW (OLD, if the file is opened with RESET).
Record length	Any positive integer	133 bytes.
Access method	DIRECT or SEQUENTIAL	SEQUENTIAL.
Record type	FIXED or VARIABLE	VARIABLE for new files; for old files, the record type is established at the file creation.
Carriage control	LIST, CARRIAGE, FORTRAN, NOCARRIAGE, or NONE	LIST for all text files; NOCARRIAGE for all other files. Old files use their existing carriage control parameter.

The following differences exist between the Version 1 and Version 2+ OPEN syntax. See the *VAX PASCAL Reference Manual* for a complete description of the current OPEN procedure.

- In Version 1, the file name is specified as a string constant (VAX/VMS file specification) or a logical name. In Version 2+, a string expression containing a file specification can be used as the file name.
- In Version 2+, the parameter values READONLY and UNKNOWN have been added to the history parameter.
- In Version 2+, the parameter value KEYED has been added to the access method parameter.
- In Version 2+, the default record type is VARIABLE for new text files and files of type FILE OF VARYING; for all other new files, the default is FIXED. The default for old files remains the same.

- In Version 2+, the default carriage control is LIST for text files and files of type FILE OF VARYING. The default for all other file types and for old files remains the same.
- Version 2+ includes six new parameters for the OPEN procedure: organization, disposition, sharing, user action, default, and error recovery. These parameters, their possible values, and their defaults are described in the *VAX PASCAL Reference Manual.*

Note that although direct access to text files is prohibited in both Version 1 and Version 2+, the point at which the error occurs differs. In Version 1, an OPEN procedure is allowed to specify direct access for a text file; the error occurs when a FIND procedure attempts to access the file. In Version 2+, an OPEN procedure that specifies direct access to a text file causes an error to be generated.

D.1.5 Specifying Qualifiers in the Source Code

In Version 1 of VAX PASCAL, you can specify compiler qualifiers within comments in the source code. VAX PASCAL Version 2+ does not support this feature. It is recommended that you specify these qualifiers with the PASCAL command when you compile the program. Alternatively, you can use attributes in your program to perform some of the same operations that are performed by compiler qualifiers. For more information, see the *VAX PASCAL Reference Manual.*

In Version 1, the CHECK qualifier (abbreviated C) generates code to perform run-time checks. The CROSS_REFERENCE qualifier (X) produces a cross-reference listing of identifiers. The DEBUG qualifier (D) generates records for the VAX/VMS Debugger. The LIST qualifier (L) produces a source listing file. The MACHINE_CODE qualifier (M) includes machine code in the source listing file. The STANDARD qualifier (S) prints informational messages indicating the use of VAX PASCAL extensions. The WARNINGS qualifier (W) prints diagnostics for warning-level errors.

The following syntax indicates how to specify qualifiers:

```
(*${qualifier},...[[; comment]] *)
```

qualifier
A qualifier name or a 1-character abbreviation.

comment
The text of a comment, which is optional.

The first character after the comment delimiter must be a dollar sign ($), which cannot be preceded by a space.

To enable a qualifier, use a plus sign (+) after the qualifier's name or abbreviation. To disable a qualifier, use a minus sign (−) after the qualifier's name or abbreviation. You can specify any number of qualifiers in a single comment. You can also include text in the comment after the qualifiers. The text must be separated from the list of qualifiers by a semicolon.

You can use qualifiers in the source code to enable and disable options during compilation. For example, to generate check code for only one procedure in a program, insert a comment that enables the CHECK qualifier before the procedure declaration. After the end of the procedure declaration, include a comment that disables the qualifier. For example:

```
(*$C+; enable CHECK for TEST1 only *)

PROCEDURE TEST1;
    .
    .
    .
END;
(*$C-; disable CHECK option *)
```

You can specify qualifiers in both the source code and the PASCAL command line. 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.

D.2 /OLD_VERSION Qualifier

The VAX PASCAL standard in Version 1 conflicts in some respects with that of Version 2+, which is based on Level 0 of the PASCAL standard. The /OLD_VERSION qualifier on the PASCAL command cause the compiler to default to the VAX PASCAL Version 1 standard when conflicts arise. By default, /OLD_VERSION is disabled so that the compilation conforms to the Pascal standard.

Because the Version 2+ compiler performs many optimizations on the source code, you should also enable the /NOOPTIMIZE qualifier during the recompilation of old programs. The /NOOPTIMIZE qualifier prevents the compiler from making optimizations that might cause an old program to behave unexpectedly when it is recompiled.

The following sections describe the conflicts between Version 1 and Version 2+ and explain how they are resolved by the /OLD_VERSION qualifier.

D.2.1 Comment Delimiters

Unlike Version 1, Version 2+ considers the opening comment delimiters (* and { equivalent; likewise, the closing delimiters *) and } are considered equivalent. Therefore, a comment begun with (* can be terminated with }, and a comment begun with { can be terminated by *).

Recompilation of the program with the /OLD_VERSION qualifier will cause the Version 1 restriction to be enforced so that you cannot combine comment delimiters in this way.

D.2.2 %INCLUDE Files

In Version 1 of VAX PASCAL, the default file type of a %INCLUDE file is DAT. However, in Version 2+, the default file type is PAS.

You must use the /OLD_VERSION qualifier to recompile a program that includes one or more files that have a file type of DAT.

D.2.3 Multidimensional Packed Arrays

Version 1 of the VAX PASCAL compiler interprets the shorthand form of the array type definition

```
PACKED ARRAY[x,y,z]
```

to be equivalent to the following longer definition:

```
ARRAY[x] OF ARRAY[y] OF PACKED ARRAY[z]
```

That is, only the last dimension of the array is packed. In Version 2+, however, the shorthand definition above is equivalent to this longer definition:

```
PACKED ARRAY[x] OF PACKED ARRAY[y] OF PACKED ARRAY[z]
```

In the Version 2+ interpretation, all dimensions of the array are packed.

You must use the /OLD_VERSION qualifier to recompile a program that includes a multidimensional packed array of which you want only the last dimension to be packed.

D.2.4 Storage of Components

In Version 1 of VAX PASCAL, a component of the subrange type 0..0 in a packed record or array is allocated one bit of storage. In Version 2+, however, a component of this type is not allocated any storage.

You must use the /OLD_VERSION qualifier to recompile a program in which one bit of storage is required to be allocated for such a component.

D.2.5 Storage of Sets

In Version 1 of VAX PASCAL, an unpacked set type is allocated 256 bits. In Version 2+, the allocation size of an unpacked set depends on the set's base type and on whether the unpacked set is allocated in a packed or an unpacked context. See the *VAX PASCAL Reference Manual* for details.

You must use the /OLD_VERSION qualifier to recompile a program in which an unpacked set requires an allocation size of 256 bits.

D.2.6 TEXT Files and FILE OF CHAR

Version 1 of VAX PASCAL considers the predefined file types TEXT and FILE OF CHAR to be equivalent. In Version 2+, however, files of type TEXT are composed of complete lines of characters terminated by an end-of-line marker, while files of type FILE OF CHAR are composed of individual characters. The *VAX PASCAL Reference Manual* provides detailed information about the differences between these two file types.

You must use the /OLD_VERSION qualifier to recompile a program that requires a TEXT file and a FILE OF CHAR to be treated identically.

D.2.7 MOD Operator

The MOD operator, as defined by Version 1 of VAX PASCAL, returns the remainder that results from the DIV operation. In Version 2+, however, the MOD operator is equivalent to the mathematical modulus operation. Therefore, Version 2+ allows you to perform the operation I MOD J only when J is a positive number; the MOD function always returns a value from 0 to J—1. To compute the remainder from the DIV operation, Version 2+ provides the REM operator. (See the *VAX PASCAL Reference Manual* for more information about the MOD and REM operators.)

You must use the /OLD_VERSION qualifier to recompile a program in which you use the MOD operator to compute the remainder.

D.2.8 String Variable Parameters to the READ Procedure

In Version 1 of VAX PASCAL, if a READ procedure encounters an end-of-line marker as the first character to be read into a string variable, it ignores the marker and advances the file position to the beginning of the next line of input. In Version 2+, a READ procedure never skips an end-of-line marker that is the first character to be read into a string variable. If a READ procedure encounters an initial end-of-line, the file remains positioned at the end of the line; you must call a READLN procedure to advance the file position to the next line. See the *VAX PASCAL Reference Manual* for further discussion of the READ procedure with string variable parameters.

You must recompile with the /OLD_VERSION qualifier to cause a READ procedure to skip one end-of-line marker that it encounters as the first character to be read into a string variable.

D.2.9 Field Widths

In Version 1 of VAX PASCAL, a value of type REAL or SINGLE is written with a default field width of 16 characters; a value of type DOUBLE, with 24. In Version 2+, however, the default field width for a value of type REAL or SINGLE is 12 characters; for a value of type DOUBLE, 20.

In addition, Version 1 of VAX PASCAL always expands the field width of a real number written in decimal format (when necessary) so that the real number is preceded by a leading blank. No leading blank is inserted in Version 2+.

You must use the /OLD_VERSION qualifier to recompile a program in which you want to use the default field width specifications of Version 1.

D.2.10 Global Identifiers

In Version 1 of VAX PASCAL, the names of program-level procedures and functions are considered global identifiers. However, in Version 2+, such names are not considered global unless they have the GLOBAL attribute.

You must use the /OLD_VERSION qualifier to recompile a program in which the names of program-level routines are to be made global by default.

D.2.11 Allocation in Program Sections

Unlike Version 1, Version 2+ of VAX PASCAL does not allocate storage for static, program-level variables in an overlaid program section. (See the *VAX PASCAL Reference Manual* for information about program section allocation.)

To cause the Version 2+ compiler to treat static, program-level variables and routine identifiers in the same manner as in Version 1, you must use the /OLD_VERSION qualifier. You can also apply the OVERLAID attribute (see the *VAX PASCAL Reference Manual*) to a compilation unit to cause the storage for its static, program-level variables to be allocated in an overlaid program section. Enabling /OLD_VERSION has the same effect as applying the OVERLAID attribute to all compilation units in a program.

D.3 Minor Language Changes

Some minor language changes that were made in Version 1 cannot be controlled by the /OLD_VERSION qualifier. Such changes, however, are not likely to have adverse affects on most existing VAX PASCAL programs. These changes are as follows:

- To flag language extensions when the /STANDARD qualifier is enabled, Version 2+ uses the Pascal standard proposed by the International Organization for Standardization (ISO). The Version 1 language is defined by the *PASCAL User Manual and Report* by Jensen and Wirth.

- In Version 2+, the /STANDARD qualifier is disabled by default. The Version 2+ compiler does not automatically flag extensions to the Pascal language definition contained in the ISO standard. /STANDARD is enabled in Version 1.
- Version 2+ ignores all comments whose first character (inside the opening delimiter) is a dollar sign ($). Note that this behavior prohibits the specification of compile-time qualifiers in the source code, which is legal in Version 1 (see Section D.1.5).
- In Version 2+, the /CHECK qualifier is enabled by default to check the bounds of array and character-string assignments. You can change the default if you wish, and you can also specify the checking of other aspects of your program. /CHECK is disabled by default in Version 1 and does not allow you to specify checking options.
- In Version 2+, a change in the value of the control variable inside the body of a FOR statement does not affect the number of times the loop body is executed. (This behavior is the reverse of Version 1.)
- Version 2+ considers the value of EOLN to be FALSE when the value of EOF is TRUE. (In Version 1, EOLN returns the value of TRUE at the end-of-file.) This change is necessary to make VAX PASCAL conform to the ISO standard, which forbids a program from testing for EOLN at the end-of-file.
- A negative field-width value in a WRITE or WRITELN procedure call generates an error in Version 2+. In Version 1, a negative field-width value defaults to 0.
- When writing double-precision values, Version 2+ output procedures indicate the exponent by the letter E. (Version 1 uses the letter D on output values.) Note, however, that input procedures in both Version 1 and Version 2+ accept either D or E as the exponent letter of double-precision values.
- Under VAX/VMS Version 4.6 or higher and Version 2+, the default length of a record in a text file is 255 bytes. Under a VAX/VMS version prior to VAX/VMS 4.6 and Version 2+, the default length of a record is 133 bytes. Because of an error in Version 1, the default length is actually 254, contrary to the description in the documentation.
- Sets in Version 2+ have allocation sizes different from those in Version 1. See the *VAX PASCAL Reference Manual* for a complete description.
- LIB$ESTABLISH, the Run-Time Library procedure that sets up condition handlers, cannot be used in Version 2+. Instead, use the new predeclared procedures ESTABLISH and REVERT.

- Run-time condition values have new values in Version 2+. These values are contained in the file SYS$LIBRARY:PASDEF.PAS. To make them available in your program, include the file in a CONST section.
- In Version 2+, when a nonlocal GOTO statement transfers control from a routine to a labeled statement in an enclosing block, any condition handlers established by intervening routines are called first with the condition SS$_UNWIND. In Version 1, a nonlocal GOTO statement transfers control directly to the labeled statement; no condition handlers are executed for intervening routines.
- In Version 2+, you cannot use the predeclared functions SNGL and ORD as function parameters using the Version 1 syntax for function parameter declarations. You must rewrite the formal parameter declarations to use the newer syntax (see the *VAX PASCAL Reference Manual*).

Appendix E

Examples of Using System Routines

The appendix contains examples that involve accessing VAX/VMS system services. It contains sections with examples of the following:

- Calling RMS procedures
- Synchronizing processes using an AST routine
- Accessing devices using synchronous I/O
- Communicating with other processes
- Sharing code and data
- Gathering and displaying data at terminals
- Creating, accessing, and ordering files
- Measuring and improving performance
- Accessing HELP libraries
- Creating and managing other processes
- Translating logical names

E.1 Calling RMS Procedures

When you call a Record Management Service, you must supply a value for each parameter. The order of the arguments must correspond with the order shown in the *Introduction to VAX/VMS System Routines.* If you omit an argument, the procedure uses a default value of zero.

Source Program Number 1

In the following program, RMS symbolic status codes are defined in the environment file STARLET.PEN, as are some of the external RMS routines declarations. STARLET.PEN contains complete formal declarations for the regular file and record processing routines. To use one of these routines in your program, you simply inherit STARLET.PEN and call the routine. The procedure name format for these routines is $procedure_name.

A few of the special RMS routines, however, are not declared in STARLET, and must be declared in your program. The procedure name format for these routines is SYS$procedure_name. The following program calls the RMS procedure SYS$SETDDIR to set the default directory for a process.

```
PROGRAM Setddir (OUTPUT);

TYPE
    Word_Integer = [WORD] 0..65535;

VAR
    Dir_Status : INTEGER;

FUNCTION SYS$SETDDIR (
    New_Dir     : [CLASS_S] PACKED ARRAY [1..u:INTEGER] OF CHAR;
    Old_Dir_Len : Word_Integer := %IMMED 0;                     ❶
    Old_Dir     : VARYING [lim2] OF CHAR := %IMMED 0):
        INTEGER; EXTERN;

BEGIN          (* main program *)

    Dir_Status := SYS$SETDDIR ( '[COURSE.PROG.PAS]');
    IF NOT ODD (Dir_Status)
        THEN WRITELN ('Error in SYS$SETDDIR call.')

END.           (* main program *)
```

❶ The $SETDIR routine accepts three parameters. To omit the second and third arguments, which are optional, supply default values in the formal declaration of the routine.

Sample Use

```
$ DIRECTORY                                                     ❷

Directory DISK$COURSE:[COURSE.PROG.PAS.CALL]

ADDIT.BAS;1     DOCOMMAND.PAS;2  GETINPUT.PAS;1  GETMSG.PAS;5
LAB1.PAS;1      LAB2.PAS;1       LAB3.PAS;1      SETDDIR.PAS;9
SHOWSUM.PAS;3

Total of 9 files.
```

```
$ PASCAL SETDDIR
$ LINK SETTDIR                                                    ❸
$ RUN SETTDIR
$
$
$ DIRECTORY                                                       ❹

Directory DISK$COURSE:[COURSE.PROG.PAS]

CALL.DIR;1         COMU.DIR;1         DEVC.DIR;1         FILE.DIR;1
HADN.DIR;1         INTR.DIR;1         MNAG.DIR;1         PERF.DIR;1
SHAR.DIR;1         SYNC.DIR;1         TERM.DIR;1         PERF.DIR;1

Total of 11 files.
```

❷ Before the program is run, the default directory is set to SYS$DISK:[COURSE.PROG.PAS.CALL] which contains the file SETDDIR.PAS.

❸ The program is compiled, linked, and executed.

❹ Another DIRECTORY command shows that the default directory has changed. The current directory is now SYS$DISK:[COURSE.PROG.PAS].

Source Program Number 2

```
[INHERIT('sys$library:starlet')]

PROGRAM rfa_read (input, output);

{++}
{
{       This is an example of using RMS RFA access
{       in a VAX PASCAL program.
{
{       This program reads a file, then prints it
{       backwards.  This is done by saving the RFAs as
{       the file is read sequentially, then using RFAs
{       to step through the file backwards.
{
{       With RFAs, the file can be positioned to any
{       known record, then read sequentially or accessed
{       randomly.
{
{       The procedures shown here are for files of type
{       FILE OF VARYING OF CHAR.
{
{       Assumptions/Restrictions:
{           File is <= 1000 records long.
{           Records are truncated to 132 characters.
{
{--}
```

```
CONST
    max_records = 1000;

TYPE
    file_type       = FILE OF VARYING [132] OF CHAR;
    file_name_type  = VARYING [255] OF CHAR;
    ptr_to_fab      = ^fab$type;
    ptr_to_rab      = ^rab$type;
    rfa_type        = RECORD
                          rfa0 : [LONG] UNSIGNED;
                          rfa4 : [WORD] 0..65535;
                      END;

[EXTERNAL]
PROCEDURE lib$stop
    (%IMMED cond_value : [LIST] INTEGER); EXTERNAL;

[EXTERNAL]
FUNCTION pas$rab
    (VAR f : [UNSAFE] file_type) : ptr_to_rab; EXTERNAL;

FUNCTION position_of
    (VAR f : file_type) : rfa_type;

    {+}
    {
    {   This function returns the current position, i.e., the
    {   RFA, of the pascal file variable parameter.
    {
    {-}

    VAR
        rab : ptr_to_rab;
        rfa : rfa_type;

    BEGIN
    rab := pas$rab( f );
    rfa.rfa0 := rab^.rab$l_rfa0;
    rfa.rfa4 := rab^.rab$w_rfa4;
    position_of := rfa;
    END;

PROCEDURE position_to
    (VAR f      : file_type;
     VAR rfa    : rfa_type);

    {+}
    {
    {   This procedure positions a pascal-file-variable
    {   to a given RFA.  It is straight forward except
    {   for the RESET, which is needed to "reset" the
    {   PASRTL internal structures when EOF is true.
    {
    {-}

    VAR
        rab     : ptr_to_rab;
        status  : INTEGER;
    BEGIN
    IF eof( f )
    THEN
            RESET( f );                  { reset EOF status }

    rab := pas$rab( f );
    rab^.rab$l_rfa0 := rfa.rfa0;         { put RFA into RAB }
    rab^.rab$w_rfa4 := rfa.rfa4;
    rab^.rab$b_rac  := rab$c_rfa;        { set for RFA access }

    status := $get( rab^ );              { do the RMS get }
    IF NOT odd( status )
    THEN
        lib$stop( status );

    rab^.rab$b_rac  := rab$c_seq;        { restore to sequential access }

    f^.length := rab^.rab$w_rsz;         { update length of VARYING string }
                                         {   usually done by PASRTL }
    END;

FUNCTION user_open
    (VAR fab : fab$type;
     VAR rab : rab$type;
     VAR f   : file_type) : INTEGER;

    {+}
    {
    {   This user-action routine modifies the FOP fields
    {   to allow non-sequential access to the file.
    {
    {-}

    VAR
        status : INTEGER;

    BEGIN
    fab.fab$v_sqo := false;              { allow non-sequential access }

    status := $open( fab );              { open file and connect RAB }
    IF odd( status )
    THEN
        status := $connect( rab );

    user_open := status;                 { return $open or $connect status }
    END;

VAR
    f           : FILE OF VARYING [132] OF CHAR;
    file_name   : file_name_type;
    i, line_num : INTEGER;
    rfa_array   : ARRAY [1..max_records] OF rfa_type;
```

```
BEGIN
WRITE( 'file-name> ' );
IF NOT eof
THEN
    BEGIN
    {+}
    {
    {   Get file-name and prepare for reading.
    {
    {-}
    READLN( file_name );

    OPEN( FILE_VARIABLE      := f,
          FILE_NAME          := file_name,
          HISTORY            := READONLY,
          USER_ACTION        := user_open );
    RESET( f );

    {+}
    {
    {   Read file saving RFAs in rfa_array.  Remember
    {   that the RESET has already caused first
    {   record to be read into file buffer variable.
    {
    {-}
    line_num := 0;
    WHILE NOT eof( f ) DO
        BEGIN
        line_num := line_num + 1;
        rfa_array[line_num] := position_of( f );
        GET( f );
        END;

    {+}
    {
    {   Print file backwards.
    {
    {-}
    FOR i := line_num DOWNTO 1 DO
        BEGIN
        position_to( f, rfa_array[i]);
        WRITELN( f^ );
        END;

    END;
END.
```

E.2 Synchronizing Processes Using an AST Routine

The methods for requesting and declaring an AST procedure are illustrated in the following example.

Source Program

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Timer_Ast
(INPUT, OUTPUT);

(*   ASTPROC.PAS                                              *)
(*   This program sets a 10 second timer which requests       *)
(*   an AST.  The main program then performs arithmetic       *)
(*   operations for the user, interrupted after 10 seconds    *)
(*   by the timer AST.                                        *)


CONST  Delay = '0 ::10.00';
TYPE  Quadword = RECORD
                 Lo : UNSIGNED;
                 Hi : INTEGER;
                 END;


VAR  Bin_Delay : Quadword;
     Ast_Output: [VOLATILE] TEXT;
     Num1, Num2, Sys_Stat: INTEGER;

(* Declare external RTL routines *)


[ASYNCHRONOUS] FUNCTION LIB$DATE_TIME( VAR Dst_Str:
    VARYING [u] OF CHAR): INTEGER; EXTERN;


[ASYNCHRONOUS] PROCEDURE LIB$STOP( %IMMED Cond_Value: INTEGER);
    EXTERN;

(* Declare AST procedure  *)

[UNBOUND,ASYNCHRONOUS] PROCEDURE Ast_Proc;                    ❶

   (*  This routine is called as an AST procedure.  It prints  *)
   (*  the current time at the terminal.                       *)

   VAR  Cur_Time :  VARYING [80] OF CHAR;
        Time_Stat : INTEGER;

   BEGIN
      Time_Stat:= LIB$DATE_TIME (Cur_Time);
      IF NOT ODD (Time_Stat) THEN LIB$STOP (Time_Stat);
      WRITELN (Ast_Output);
      WRITELN (Ast_Output,'   The time is now:   ', Cur_Time);❹
      WRITELN (Ast_Output)
   END;  (* Ast_Proc *)


(* Begin main program  *)

BEGIN
   OPEN (Ast_Output, 'TT:');(* output channel for Ast_Proc *)❷
   REWRITE (Ast_Output);

   (* Convert delay interval to binary format and set timer *)
   Sys_Stat := $BINTIM (Delay, Bin_Delay);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Sys_Stat:= $SETIMR (Daytim:= Bin_Delay, Astadr:= Ast_Proc);❸
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Prompt user for two numbers and multiply them *)
   REPEAT
      WRITELN ('Enter two integers to be multiplied.');
      WRITELN ('(Enter two zeros to quit.)');
      READLN (Num1, Num2);
      WRITELN (' The product is: ', Num1 * Num2)
      UNTIL (Num1 = 0) AND (Num2 = 0);
   WRITELN (' Arithmetic operations completed.')
END.
```

Sample Use

```
$ RUN ASTPROC
Enter two integers to be multiplied.
(Enter two zeros to quit.)
12
12
 The product is:         144
Enter two integers to be multiplied.
(Enter two zeros to quit.)

   The time is now:11-OCT-1984  16:18:43.54                ❹
23
45
 The product is:        1035
Enter two integers to be multiplied.
(Enter two zeros to quit.)
0
0
 The product is:           0
 Arithmetic operations completed.
$
```

❶ AST_PROC must be declared UNBOUND, because it is passed to $SETIMR using %IMMED.

❷ The AST routine needs an output channel to the terminal, separate from the channel used by the main program. The channel is opened and associated to the terminal (TT:) in the main program, because this is a time-consuming operation. You should avoid including unnecessary lengthy operations in AST routines.

Any global variables, like AST_OUTPUT, referenced by a routine declared ASYNCHRONOUS, must be declared VOLATILE. This attribute informs the compiler that the variable is subject to change at any time, and should be optimized carefully.

❸ The ASTADR parameter in the call to $SETIMR is the address of the entry point of an AST routine. The AST routine will be executed when the timer expires. If the AST routine is to be executed repeatedly, rather than just once, the timer should be reset at the end of the AST routine.

❹ When the AST is delivered, the AST routine interrupts the program and outputs this message.

E.3 Accessing Devices Using Synchronous INPUT/OUTPUT

The following example performs output to a terminal via the $QIOW system service.

Source Program

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Qiow( OUTPUT);

(*   QIOW.PAS                                               *)
(*   This program illustrates the use of the $QIOW system   *)
(*   service to perform synchronous I/O to a terminal.      *)

CONST  Text_String = 'This is from a $QIOW.';
       Terminal = 'TT:';

TYPE   Word_Integer = [WORD] 0..65535;
       Io_Block = RECORD
                    Io_Stat, Count: Word_Integer;
                    Dev_Info : INTEGER
                    END;

VAR  Term_chan : Word_Integer;
     Counter   : INTEGER;
     Sys_Stat  : INTEGER;
     Iostat_Block : Io_Block;

(* Declare external RTL routine *)

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

BEGIN
   (* Assign the channel number *)
   Sys_Stat := $ASSIGN (Terminal, Term_Chan,,);                ❶
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Output the message twice *)
   FOR Counter:= 1 TO 2 DO
      BEGIN
         Sys_Stat:= $QIOW (Chan:= Term_Chan,
                           Func:= IO$_WRITEVBLK,
                           Iosb:= Iostat_Block,                 ❷
                             P1:= Text_String,
                             P2:= LENGTH(Text_String),
                             P4:= 32);
         IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
         IF NOT ODD (Iostat_Block.Io_Stat) THEN
                    LIB$STOP (Iostat_Block.Io_Stat);

      END    (* for *)
END.
```

Sample Use

```
$ RUN QIOW
This is from a $QIOW.
This is from a $QIOW.
$
```

❶ TERM_CHAN receives the channel number from the $ASSIGN system service.

The process permanent logical name SYS$OUTPUT is assigned to your terminal when you login. The $ASSIGN system service will translate the logical name to the actual device name.

❷ The function IO$_WRITEVBLK requires values for parameters P1, P2, and P4. P1 is the starting address of the buffer containing the message. P2 is the number of bytes to be written to the terminal (in this example, 21). P4 is the carriage control specifier; 32 indicates single space carriage control.

$QIOW is used, instead of $QIO, to insure that the output operation will complete before the program terminates.

E.4 Communicating with Other Processes

The following example shows how to create a global section and use it to transfer data between two processes. This example requires that you run programs GLOBAL1 and GLOBAL2 in the same UIC group. GLOBAL1 creates and maps the section, but GLOBAL2 only maps the section. Therefore, GLOBAL1 must be run first.

Source Program 1

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Global1(OUTPUT);

(*   GLOBAL1.PAS                                        *)
(*   This program creates and maps a global section.    *)
(*   Data in the section file is accessed through an    *)
(*   array.                                             *)


TYPE Int_Pointer = ^INTEGER;              (* for addresses *)
     File_Type = FILE OF INTEGER;
     Word_Integer = [WORD] 0..65535;


VAR  My_Adr, Sys_Adr: ARRAY [1..2] OF Int_Pointer;        ❶
     Iarray: [VOLATILE,ALIGNED(9),BYTE(512)] ARRAY [1..50] OF INTEGER;
     Sec_Chan:  UNSIGNED;
     Name:      PACKED ARRAY [1..4] OF CHAR;
     Sec_File:  File_Type;
     Sec_Flags: INTEGER;
     Sys_Stat, Counter: INTEGER;


(* Declare user_action open function  *)

FUNCTION Get_Chan (
     VAR File_Fab: FAB$TYPE;  VAR File_Rab: RAB$TYPE;
     VAR A_File: File_Type): INTEGER;

   (* This routine is called as a user_action function  *)
   (* by the OPEN statement.  The channel number for    *)
   (* file is stored in the global variable SEC_CHAN.   *)

   VAR  Open_Status: INTEGER;

   BEGIN
   (* Define appropriate FAB fields *)
   File_Fab.FAB$V_CBT := FALSE;
   File_Fab.FAB$V_CTG := TRUE;
   File_Fab.FAB$V_UFO := TRUE;
   File_Fab.FAB$L_ALQ := 1;

   (* Open the file *)
   Open_Status:= $CREATE (Fab:=File_Fab);
   IF ODD (Open_Status)
     THEN
         BEGIN
         (* get channel number *)
         Sec_Chan:= File_Fab.FAB$L_STV                     ❸
         END;
   Get_Chan:= Open_Status
   END;      (*  get_chan  *)


(* Declare external RTL routine  *)


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

```
BEGIN
   (* Associate with common cluster MYCLUS *)
   $ASCEFC (Efn:= 64, Name:= 'MYCLUS');

   (* Open the file *)                                          ❷
   OPEN (File_Variable:= Sec_File, File_Name:= 'SECTION.DAT',
         History:= New, User_Action:= Get_Chan);

   (* Obtain starting and ending addresses of the section *)
   My_Adr[1]:= ADDRESS (Iarray[1]);
   My_Adr[2]:= ADDRESS (Iarray[50]);

   (* Create and map the temporary global section *)            ❹
   Name:= 'GSEC';
   Sec_Flags:= SEC$M_GBL + SEC$M_WRT + SEC$M_DZRO;
   Sys_Stat:= $CRMPSC (My_Adr, Sys_Adr,, Sec_flags, Name,,,
                       %IMMED Sec_Chan, 1,,,);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Write data to the global section *)
   FOR Counter:= 1 TO 50 DO
      Iarray[Counter]:= Counter;

   (* Write the pages to the file *)
   Sys_Stat := $UPDSEC (Inadr:= Sys_Adr, Efn:= 1);              ❺
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Wait until pages have been written back to the disk *)
   Sys_Stat := $WAITFR (Efn:= 1);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Wait for GLOBAL2 to update the section *)
   Sys_Stat:= $SETEF (Efn:= 72);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('Waiting for GLOBAL2 to update section.');
   Sys_Stat:= $WAITFR (Efn:= 73);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Print the modified pages *)
   WRITELN( 'Modified data in the global section:');
   FOR Counter:= 1 TO 50 DO
      BEGIN
         WRITE (Iarray[Counter]:5);
         IF (Counter REM 10) = 0
             THEN WRITELN;
      END
END.
```

Source Program 2

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Global2( OUTPUT);

(*   GLOBAL2.PAS                                              *)
(*   This program maps and modifies a global section          *)
(*   after GLOBAL1 creates it.  Programs GLOBAL1 and          *)
(*   GLOBAL2 synchronize the processing of the global         *)
(*   section via common event flags.                          *)
```

```
TYPE Int_Pointer = ^INTEGER;           (* for addresses *)
     File_Type = FILE OF INTEGER;

VAR  My_Adr: ARRAY [1..2] OF Int_Pointer;                 ❶
     Iarray: [VOLATILE, ALIGNED(9),BYTE(512)] ARRAY [1..50] OF INTEGER;
     Sys_Stat, counter: INTEGER;

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

BEGIN
   (* Obtain starting and ending addresses of the section *)
   My_Adr[1]:= ADDRESS (Iarray[1] );
   My_Adr[2]:= ADDRESS (Iarray[50] );

   (* Associate with common cluster MYCLUS, and wait for   *)
   (* event flag to be set.                                *)
   Sys_Stat:= $ASCEFC (Efn:= 64, Name:= 'MYCLUS' );
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Sys_Stat:= $WAITFR (Efn:= 72);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Map global section, using WRITE flag  *)            ❻
   sys_Stat:= $MGBLSC (Inadr:= My_Adr, Flags:= SEC$M_WRT,
                       Gsdnam:= 'GSEC');
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

   (* Output data in global section, and double each value *)
   WRITELN ('Original data in global section:');
   FOR Counter:= 1 TO 50 DO
      BEGIN
         WRITE (Iarray[Counter]:5);
         IF (Counter REM 10) = 0
            THEN WRITELN;
            Iarray[Counter]:= Iarray[Counter] * 2
      END;

   (* Set event flag to allow GLOBAL1 to continue *)
   Sys_Stat:= $SETEF (Efn:= 73);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
END.
```

Sample Use

```
$ ! FROM ONE PROCESS
$ RUN GLOBAL1
Waiting for GLOBAL2 to update section.
Modified data in the global section:
    2    4    6    8   10   12   14   16   18   20
   22   24   26   28   30   32   34   36   38   40
   42   44   46   48   50   52   54   56   58   60
   62   64   66   68   70   72   74   76   78   80
   82   84   86   88   90   92   94   96   98  100
```

```
$
$
$
$ ! FROM ANOTHER PROCESS
$ RUN GLOBAL2
Original data in global section:
    1     2     3     4     5     6     7     8     9    10
   11    12    13    14    15    16    17    18    19    20
   21    22    23    24    25    26    27    28    29    30
   31    32    33    34    35    36    37    38    39    40
   41    42    43    44    45    46    47    48    49    50
$
```

❶ The $CRMPSC system service maps pages starting at page boundaries. The ALIGN attribute, with a value of 9, is used to ensure that IARRAY starts on a page boundary. (If the binary address of the first element of IARRAY ends in 9 zeros, then it is page aligned.)

❷ In order to map a disk file as a section, we need a channel for the file. A channel is assigned by the OPEN statement, and the user-action feature of the OPEN statement allows you to obtain the number of the assigned channel.

❸ Because the user-file-open option (FAB$V_UFO) was specified in the file FAB, the file channel number is stored in the file FAB at offset FAB$L_STV.

❹ The starting and ending process virtual addresses of the section are placed in MY_ADR. The output argument SYS_ADR receives the starting and ending system virtual addresses. The flag SEC$M_GLOBAL requests a global section. The flag SEC$M_WRT indicates that the pages should be writeable as well as readable. The SEC$M_DZRO flag requests pages filled with zeros.

❺ Data is placed in the global section and then the section is updated with the $UPDSEC system service. The $UPDSEC system service executes asynchronously. Therefore, event flag 1 is used to synchronize completion of the update operation.

❻ GLOBAL2 maps the existing section as writeable by specifying the SEC$M_WRT flag. Note that the SEC$M_DZRO flag is not set, because that would destroy any data that might be in the section.

E.5 Sharing Code and Data

The program called SHAREDFIL is used to update records in a relative file. SHARING := READWRITE is specified on the OPEN statement to invoke the RMS file sharing facility. In this example, the same program is used to access the file from two processes.

Source Program

```
PROGRAM Sharedfil (INPUT, OUTPUT);

(*   SHAREDFIL.PAS                                     *)❶
(*   This program can be run from two or more processes *)
(*   to illustrate the use of an RMS shared file to     *)
(*   share data.  The program requires the existence of *)
(*   a relative file named REL.DAT.                     *)

CONST  Prompt= 'Record number (CTRL/Z to quit): ';
       %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS'                ❷

TYPE Char_Array = PACKED ARRAY [1..10] OF CHAR;

VAR  Rel_File: FILE OF Char_Array;
     Rec_Num : INTEGER;

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

PROCEDURE Process_Rec;
   (* Output a record and modify value, if necessary *)
   (* global variables: rec_num, rel_file           *)

   BEGIN
      WRITELN( 'file record is: ', Rel_File^);
      (* Request updated record *)
      WRITE( 'New Value or CR: ');
      IF NOT EOLN( INPUT)
         THEN
            BEGIN                   (*  Read new information from SYS$INPUT *)
                                    (*  directly into the file buffer of    *)

            READLN (Rel_File^);
            UPDATE (Rel_File);
            IF STATUS (Rel_File) <> 0
               THEN LIB$STOP(%x218004 + ((STATUS(Rel_File) + 200) * 8))
            END   (* if not <CR> *)
         ELSE READLN  (* read past the <CR> *)
   END;  (* process_rec *)
```

```
BEGIN   (* main *)
   OPEN (FILE_VARIABLE:= Rel_File, FILE_NAME:= 'REL.DAT',
         HISTORY:= Old,            ACCESS_METHOD:= Direct,
         ORGANIZATION:= Relative,  SHARING:= Readwrite);      3

   WRITE (Prompt);
   WHILE NOT EOF (Input) DO
      BEGIN
      (* Get record *)
      READLN (Rec_Num);
      FIND (Rel_File, Rec_Num, ERROR:= CONTINUE);

      CASE STATUS (Rel_File) OF
         PAS$K_SUCCESS: IF NOT UFB (Rel_File)
               THEN Process_Rec
               ELSE WRITELN ('Did not find record ',Rec_Num);

         PAS$K_FAIGETLOC:  WRITELN ('Record locked.');

         PAS$K_ERRDURFIN:  WRITELN ('Error during FIND.');

         OTHERWISE  (* signal the error *)
           LIB$STOP(%x218004 + ((STATUS(Rel_File) + 200) * 8))


         END;   (* end case *)
      WRITE (Prompt)
      END  (* while *)
END.
```

Sample Use

```
$ PASCAL SHAREDFIL
$ LINK SHAREDFIL
$ RUN SHAREDFIL
Record number (CTRL/Z to quit): 2
file record is: MSPIGGY
New Value or CR: FOZZIE
Record number (CTRL/Z to quit): 1
file record is: KERMIT
New Value or CR:
Record number (CTRL/Z to quit): CTRL/Z
```

```
$
$
$ RUN SHAREDFIL
Record number (CTRL/Z to quit): 2                        ❹
Record locked.
Record number (CTRL/Z to quit): 2
Record locked.
Record number (CTRL/Z to quit): 2
file record is: FOZZIE                                   ❺
New Value or CR: MSPIGGY
Record number (CTRL/Z to quit): CTRL/Z
$
```

❶ This program requires a relative file named REL.DAT.

❷ The PASSTATUS.PAS file is included to define the symbolic status codes for the conditions detected by the STATUS function. Notice that no semicolon is necessary to end the INCLUDE clause.

❸ SHARING := READWRITE is specified on the OPEN statement to indicate that the file can be shared. Because manual locking was not specified, RMS will automatically control access to the file.

❹ The second process is not allowed to access record number 2 while the first process is accessing it.

❺ Once the first process has finished with record number 2, the second process can update it.

E.6 Gathering and Displaying Data at Terminals

The following example illustrates how to use SMG$ screen management routines from VAX PASCAL. It clears the screen, then divides it into two regions. In the top half of the screen, data is entered and echoed; in the bottom half of the screen, the data is echoed again.

Source Program 1

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Smg_Example;

{+}
{
{   SMG_EXAMPLE.PAS
{   This program shows the use of the RTL Screen Management
{   Package, SMG.
{
{-}
```

```
TYPE
    Cond_Code           = UNSIGNED;
    Mask                = UNSIGNED;
    Display_Id_Type     = UNSIGNED;
    Pasteboard_Id_Type  = UNSIGNED;
    Keyboard_Id_Type    = UNSIGNED;
    Key_Table_Id_Type   = UNSIGNED;
    Unsigned_Byte       = [BYTE] 0..255;
    Unsigned_Word       = [WORD] 0..65535;

PROCEDURE LIB$STOP (%IMMED Cond_Val : Cond_Code); EXTERNAL;

FUNCTION SMG$CREATE_KEY_TABLE
        (%REF New_Key_Table_Id  : Key_Table_Id_Type)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$CREATE_PASTEBOARD
        (%REF   New_Pasteboard_Id
                                : Pasteboard_Id_Type;
         %STDESCR
                Output_Device   : [TRUNCATE] PACKED ARRAY [l2..u2:INTEGER]
                                             OF CHAR := %IMMED 0;
         %REF   Pb_Rows         : [TRUNCATE] INTEGER := %IMMED 0;
         %REF   Pb_Columns      : [TRUNCATE] INTEGER := %IMMED 0;
         %REF   Preserve_Screen : [TRUNCATE] mask)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$CREATE_VIRTUAL_DISPLAY
        (%REF Num_Rows           : INTEGER;
         %REF Num_Columns        : INTEGER;
         %REF New_Display_Id     : display_id_type;
         %REF Display_Attributes : [TRUNCATE] UNSIGNED := %IMMED 0;
         %REF Video_Attributes   : [TRUNCATE] mask := %IMMED 0;
         %REF Char_Set           : [TRUNCATE] UNSIGNED := %IMMED 0)
         : Cond_Code; EXTERNAL;

FUNCTION SMG$CREATE_VIRTUAL_KEYBOARD
        (%REF   New_Keyboard_Id : Keyboard_Id_Type;
         %DESCR Filespec        : [TRUNCATE] PACKED ARRAY [l2..u2:INTEGER]
                                             OF CHAR := %IMMED 0;
         %DESCR Default_Filespec: [TRUNCATE] PACKED ARRAY [l3..u3:INTEGER]
                                             OF CHAR := %IMMED 0;
         %DESCR Resultant_Filespec
                                : [TRUNCATE] PACKED ARRAY [l4..u4:INTEGER]
                                             OF CHAR := %IMMED 0;
         %REF   Recall_Size     : [TRUNCATE] unsigned_byte := %IMMED 0)
        : Cond_Code; EXTERNAL;
```

```
FUNCTION SMG$LABEL_BORDER
        (%REF   Display_Id      : Display_Id_Type;
         %DESCR Label_Text      : [TRUNCATE] PACKED ARRAY [l2..u2:INTEGER]
                                              OF CHAR := %IMMED 0;
         %REF   Position        : [TRUNCATE] UNSIGNED := %IMMED 0;
         %REF   Units           : [TRUNCATE] INTEGER := %IMMED 0;
         %REF   Rendition_Set   : [TRUNCATE] Mask := %IMMED 0;
         %REF   Rendition_Complement
                                : [TRUNCATE] Mask := %IMMED 0;
         %REF   Char_Set        : [TRUNCATE] UNSIGNED := %IMMED 0)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$PASTE_VIRTUAL_DISPLAY
        (%REF Display_Id        : Display_Id_Type;
         %REF Pasteboard_Id     : Pasteboard_Id_Type;
         %REF Pasteboard_Row    : INTEGER;
         %REF Pasteboard_Column : INTEGER;
         %REF Top_Display_Id    : [TRUNCATE] Display_Id_Type := %IMMED 0)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$PUT_LINE
        (%REF   Display_Id      : Display_Id_Type;
         %DESCR Text            : VARYING [u2] OF CHAR;
         %REF   Line_Advance    : [TRUNCATE] INTEGER := %IMMED 0;
         %REF   Rendition_Set   : [TRUNCATE] Mask := %IMMED 0;
         %REF   Rendition_Complement
                                : [TRUNCATE] Mask := %IMMED 0;
         %REF   Wrap_Flag       : [TRUNCATE] Mask := %IMMED 0;
         %REF   Char_Set        : [TRUNCATE] UNSIGNED:= %IMMED 0;
         %REF   Direction       : [TRUNCATE] UNSIGNED := %IMMED 0)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$READ_COMPOSED_LINE
        (%REF   Keyboard_id     : Keyboard_Id_Type;
         %REF   Key_Table_Id    : Key_Table_Id_Type;
         %DESCR Received_Text   : VARYING [u3] OF CHAR;
         %DESCR Prompt_String   : [TRUNCATE] PACKED ARRAY [l4..u4:INTEGER]
                                              OF CHAR := %IMMED 0;
         %REF   Received_Len    : [TRUNCATE] Unsigned_Word := %IMMED 0;
         %REF   Display_Id      : [TRUNCATE] Display_Id_Type := %IMMED 0;
         %REF   Function_Keys   : [TRUNCATE] UNSIGNED := %IMMED 0;
         %DESCR Ini_String      : [TRUNCATE] PACKED ARRAY [l8..u8:INTEGER]
                                              OF CHAR := %IMMED 0;
         %REF   Timeout         : [TRUNCATE] INTEGER := %IMMED 0;
         %REF   Rendition_Set   : [TRUNCATE] mask := %IMMED 0;
         %REF   Rendition_Complement
                                : [TRUNCATE] Mask := %IMMED 0;
         %REF   Terminator_Code : [TRUNCATE] Unsigned_Word := %IMMED 0)
        : Cond_Code; EXTERNAL;

FUNCTION SMG$SET_PHYSICAL_CURSOR
        (%REF   Pasteboard_Id   : Pasteboard_Id_Type;
         %REF   Pb_Row          : INTEGER;
         %REF   Pb_Column       : INTEGER)
        : Cond_Code; EXTERNAL;
```

```
VAR
    Stat            : Cond_Code;
    Text            : VARYING [132] OF CHAR;
    Keyboard_Id     : Keyboard_Id_Type;
    Pasteboard_Id   : Pasteboard_Id_Type;
    Key_Table_Id    : Key_Table_Id_Type;
    Display_Id_1    : Display_Id_Type;
    Display_Id_2    : Display_Id_Type;
    Pb_Rows         : INTEGER;
    Pb_Columns      : INTEGER;
    Display_Rows    : INTEGER;

BEGIN

{+}
{
{   Create the basic data-structures needed by SMG;
{   the PASTEBOARD, KEYBOARD, and KEY-TABLE.
{
{-}

STAT := SMG$CREATE_PASTEBOARD( pasteboard_id,                           ❶
                                pb_rows    := pb_rows,
                                pb_columns := pb_columns );
IF NOT ODD( stat ) THEN lib$stop( stat );

Stat := SMG$CREATE_VIRTUAL_KEYBOARD( Keyboard_Id );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$CREATE_KEY_TABLE( Key_Table_Id );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

{+}
{
{   Create and paste two VIRTUAL-DISPLAYs; one for top half of
{   screen and another for bottom half of screen.  Label
{   the bottom display with "ECHO"
{
{-}

Display_Rows := (Pb_Rows-4) DIV 2;                                    ❷

Stat := SMG$CREATE_VIRTUAL_DISPLAY( Display_Rows, Pb_Columns, Display_Id_1 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$CREATE_VIRTUAL_DISPLAY( Display_Rows, Pb_Columns, Display_Id_2 );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$PASTE_VIRTUAL_DISPLAY( Display_Id_1, Pasteboard_Id, 1, 1);
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$PASTE_VIRTUAL_DISPLAY( Display_Id_2, Pasteboard_Id,
                                    Pb_Rows DIV 2, 1);
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );

Stat := SMG$LABEL_BORDER( Display_Id_2, 'ECHO' );
IF NOT ODD( Stat ) THEN LIB$STOP( Stat );
```

```
{+}
{
{   Read (and echo) lines in top part of screen and also echo
{   in bottom half of screen.
{
{-}

WHILE ODD( SMG$READ_COMPOSED_LINE( Keyboard_Id, Key_Table_Id,          ❸
                                   Text, '> ', , Display_Id_1)) DO
    BEGIN
    Stat := SMG$PUT_LINE( Display_Id_2, Text );
    IF NOT ODD( Stat ) THEN LIB$STOP( Stat );
    END;

{ move cursor to bottom of screen before exiting }                     ❹
SMG$SET_PHYSICAL_CURSOR( Pasteboard_Id, Pb_Rows-1, 1 );

END.
```

❶ The CREATE_PASTEBOARD routine erases the screen by default. It also returns the size of the output device which is used to create the virtual displays.

❷ The virtual displays are created and pasted onto the pasteboard.

❸ This loop consisting of calls to READ_COMPOSED_LINE and PUT_LINE cause the data to be echoed in both parts of the screen. READ_COMPOSED_LINE transparently implements multi-line recall similar to the multi-line command recall available in DCL.

❹ The call to SET_PHYSICAL_CURSOR leaves the cursor positioned at the bottom of the screen when the program exits.

Source Program 2

This program illustrates the use of the $QIO system service to communicate with the terminal, and to respond to a CTRL/C typed by a user. It also illustrates the use of function code modifiers.

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Controlc
(INPUT, OUTPUT);

(*   CONTROLC.PAS                                  *)
(*   This program illustrates the use of $QIOs to the   *)
(*   terminal and establishing an AST routine to        *)
(*   handle CTRL/C's.                              *)


TYPE Word_Integer = [WORD] 0..65535;
     Io_Status = RECORD
                 Io_Stat, Count: Word_Integer;
                 Device_Info : INTEGER
                 END;
```

```
VAR  Prompt:       PACKED ARRAY [1..48] OF CHAR;
     Buffer:       VARYING [80] OF CHAR;
     Tt_Chan:      [VOLATILE] Word_Integer;
     Ast_Output:   [VOLATILE] TEXT;
     Io_Func:      Word_Integer;
     Iostat_Block: [VOLATILE] Io_Status;
     Counter:      INTEGER;
     Sys_Stat:     INTEGER;

[ASYNCHRONOUS] PROCEDURE LIB$STOP (                         ❶
                          %IMMED Cond_Value: INTEGER); EXTERN;

[ASYNCHRONOUS, UNBOUND] PROCEDURE Cast (Channel: Word_Integer);
   (*  This procedure is called as a CTRL/C ast routine. *)

   VAR  Cancel_Stat:  INTEGER;

   BEGIN
      (* Cancel all I/O on terminal channel *)
      Cancel_Stat:= $CANCEL (Channel);                       ❼
      IF NOT ODD (Cancel_Stat) THEN LIB$STOP (Cancel_Stat);

      (* Tell user that CTRL/C AST routine was entered *)
      WRITELN (Ast_Output, 'You typed a CTRL/C')             ❽
   END;    (*  procedure cast *)


(* Begin main program  *)

BEGIN
   OPEN (Ast_Output, 'TT:');     (* output channel for cast *)
   REWRITE (Ast_Output);
   Prompt:='You have 5 seconds to enter data, or type CTRL/C';

   (* Assign channel to terminal *)
   Sys_Stat:= $ASSIGN (Devnam:= 'SYS$COMMAND', Chan:= Tt_Chan);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   (* Enable CTRL/C AST *)
   Io_Func:= IO$_SETMODE + IO$M_CTRLCAST;                    ❸
   Sys_Stat:= $QIOW (Chan:= Tt_Chan, Func:= Io_Func,
                     Iosb:=Iostat_Block,
                     P1:= %IMMED Cast, P2:= %REF Tt_Chan);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   IF NOT ODD (Iostat_Block.Io_Stat) THEN
               LIB$STOP (Iostat_Block.Io_Stat);

   (* Issue a timed read, without echo, with a prompt *)
   Io_Func:= IO$_READPROMPT + IO$M_NOECHO + IO$M_TIMED;      ❹
   Sys_Stat:= $QIOW (,Tt_Chan, Io_Func, Iostat_Block,,,
                     Buffer.Body, SIZE (Buffer.Body), 5,,
                     %REF Prompt, SIZE (Prompt));
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   Buffer.Length:= Iostat_Block.Count;                       ❺
   (* Check status of I/O and issue appropriate message *)
   IF Iostat_Block.Io_Stat = SS$_NORMAL
      THEN WRITELN ('You typed: ', Buffer)
      ELSE
         IF Iostat_Block.Io_Stat = SS$_CONTROLC
            THEN WRITELN ('Request was canceled')        ❻
            ELSE
              IF Iostat_Block.Io_Stat = SS$_TIMEOUT
                  THEN WRITELN ('You did not respond in time')
                  ELSE WRITELN ('Unexpected status: ',
                                Iostat_Block.Io_Stat)

END.
```

Sample Use

```
$ RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
You typed: HI THERE
$ RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
 CTRL/C 

You typed a CTRL/C
Request was canceled
$ RUN CONTROLC
You have 5 seconds to enter data, or type CTRL/C
You did not respond in time
$
```

❶ Because CAST is called asynchronously:

- The procedure must be declared ASYNCHRONOUS and UNBOUND.
- LIB$STOP, which is called from the procedure, must also be declared ASYNCHRONOUS.
- Tt_Chan must be declared VOLATILE.
- A separate output channel to the terminal (ast_output) must be established.

❷ Before any $QIO can be issued, a channel must be assigned to the device.

❸ A CTRL/C AST routine is declared.

- Note that this routine will only be entered for the first CTRL/C the user types.

- The formal parameter for P1 specifies that it will be passed %REF. In this case, you are passing the entry point for a routine, which should be passed %IMMED. The foreign mechanism specifier is used on the actual parameter for P1 to override the formal definition.
- The %REF mechanism specifier is used on the actual parameter for P2 to override the formal definition of %IMMED. The CAST procedure expects this parameter to be passed to it by reference.

❹ This request prompts a user for input. It sets up a time limit for waiting between characters typed (five seconds). Also, any characters typed will not be echoed on the terminal. An I/O status block is established to contain the final I/O status and byte count for characters read.

The %REF specifier is used on the actual P5 parameter to override type checking against the formal parameter definition, and to pass the parameter by reference rather than %IMMED.

❺ The LENGTH field of the varying string BUFFER is filled in manually because it is not done by the system service. System services can accept (and manipulate) only fixed-length strings.

BUFFER could be defined as a fixed-length string, but you would have to know (at compile-time) the exact size of the string to be read by the $QIOW.

❻ Specific tests are made for CTRL/C or timeout detection using the final I/O status in the I/O Status Block.

❼ The CTRL/C routine cancels any outstanding requests on the terminal channel.

❽ Before exiting, the CTRL/C AST routine issues a message to the user, indicating that the AST routine was called.

The sample run illustrates the three possibilities that may occur when running this program (entering data, not responding in time, and typing a CTRL/C).

E.7 Creating, Accessing, and Ordering Files

The following examples demonstrate how to access files using VAX RMS, VAX PASCAL file I/O routines, and run-time library routines.

Source Program 1

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Timeout
(INPUT, OUTPUT);

(*   TIMEOUT.PAS                                       *)
(*   This program illustrates the use of a user-action *)
(*   routine to initialize the I/O time-out period for *)
(*   a terminal. It also illustrates the use of the    *)
(*   STATUS and PAS$RAB functions to obtain the status *)
(*   I/O operations.                                   *)

CONST  %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS/NOLIST'              ❶

VAR    Term_File: TEXT;
       Number   : INTEGER;
       Prompt   : PACKED ARRAY [1..33] OF CHAR
                := 'Enter an integer (zero to quit): ';

PROCEDURE LIB$STOP (%IMMED Cond_Value: INTEGER); EXTERNAL;

(* Declare user_action open function  *)
FUNCTION Set_Timeout (                                         ❷
     VAR File_Fab: FAB$TYPE;  VAR File_Rab: RAB$TYPE;
     VAR A_File  : TEXT): INTEGER;

   (* This routine is called as a user_action function  *)
   (* by the OPEN statement.  The file is opened with   *)
   (* an input time-out period of 10 seconds.           *)

   VAR  Open_Status: INTEGER;

   BEGIN
      (* Set up RAB for TMO of 10 seconds *)                   ❹
      File_Rab.RAB$B_TMO:= 10;
      File_Rab.RAB$V_TMO:= TRUE;

      (* Set up RAB for RMS prompting *)
      File_Rab.RAB$V_PMT:= TRUE;                               ❺
      File_Rab.RAB$L_PBF:= IADDRESS (Prompt);
      File_Rab.RAB$B_PSZ:= SIZE (Prompt);
      (* Open the file and connect the record stream *)
      Open_Status:= $CREATE (Fab:=File_Fab);                  ❻
      IF ODD (Open_Status)
         THEN $CONNECT (Rab:=File_Rab);
      Set_Timeout:= Open_Status
   END;      (*  set_timeout  *)


PROCEDURE Process_Get_Error;

   (* Determines the RMS error that caused a PAS$K_ERRDURGET *)
   (* Global variable:  term_file *)

   TYPE Rab_Ptr = ^RAB$TYPE;
   VAR  Rab_Start: Rab_Ptr;

   FUNCTION PAS$RAB (VAR F: [UNSAFE] TEXT): Rab_Ptr; EXTERN;

   PROCEDURE LIB$SIGNAL (Sigargs:
                         [LIST,UNSAFE,IMMEDIATE] INTEGER); EXTERN;

   BEGIN  (* process_get_error *)
      Rab_Start:= PAS$RAB (Term_File);
      IF Rab_Start^.RAB$L_STS = RMS$_TMO                      ❾
         THEN WRITELN ('No response in 10 seconds...')
         ELSE LIB$SIGNAL (Rab_Start^.RAB$L_STS,
                          Rab_Start^.RAB$L_STV)
   END;  (* procedure *)


(* Begin main program  *)

BEGIN
   OPEN (File_Variable:= Term_File, File_Name:= 'TT:',        ❸
         History:= New, User_Action:= Set_Timeout);
   RESET (Term_File);

   (* Accept input until zero entered *)
   Number:= 1;
   WHILE (Number <> 0) DO
      BEGIN
         READ (Term_File, Number, ERROR:= CONTINUE);          ❼

         (* Display input and test for errors *)
         CASE STATUS (Term_File) OF                           ❽
            PAS$K_SUCCESS:
               WRITELN ('You entered: ', Number);
            PAS$K_ERRDURGET:
               Process_Get_Error;
            OTHERWISE
               LIB$STOP (%x218004 + ((STATUS(Rel_File) + 200) * 8))


            END;  (* case *)
```

```
        (* clear the rest of the line *)
        WHILE NOT EOLN (Term_File)  DO
          GET (Term_File);
      END;   (* while not zero *)
END.
```

Sample Use

```
$ PASCAL TIMEOUT
$ LINK TIMEOUT
$ RUN TIMEOUT
Enter an integer (zero to quit): 10
You entered:         10
Enter an integer (zero to quit):
No response in 10 seconds...
Enter an integer (zero to quit):  0
You entered:          0
$
```

❶ Including PASSTATUS.PAS defines the VAX PASCAL error codes detected by the STATUS and STATUSV functions.

❷ Three parameters are passed to a user action routine: the FAB, the RAB, and the file variable. Recall that FAB$TYPE and RAB$TYPE are defined in STARLET.PEN.

❸ All relevant parameters are included in the OPEN statement. The user action routine only alters RMS fields that could not be specified in the OPEN statement.

❹ The appropriate RAB fields are set to define a timeout period.

❺ When a TEXT file is opened with a user action routine, the usual VAX PASCAL prompting feature is not enabled. In other words, although read and write operations are going to the same device, in this example, the following two statements do not display the prompt at the terminal:

```
WRITE (Prompt);
READLN (Term_File, Number);
```

Instead, RMS prompting is enabled for the file by setting the appropriate field in the RAB. Now any READ or GET operation on the file translates to a "read with prompt."

❻ Once the appropriate fields are set, the file is created and a record stream is connected.

❼ The program accesses the file using a normal READ statement. When the ERROR parameter is used, it must be specified using keyword calling syntax. In addition, it must be the last parameter in the list.

❽ The STATUS function returns the status code of the last operation on the file. If the status code is PAS$K_ERRDURGET, procedure Process_Get_Error is called.

❾ VAX PASCAL only informs the program that an error occurred during the READ (or GET). To determine the actual RMS error code, procedure Process_Get_Error calls PAS$RAB to obtain a pointer to the file $RAB. The RAB$L_STS filed in the RAB contains the actual RMS status code.

If the RMS status code is RMS$TMO (timeout), the program informs the user. Any other error is unexpected, so the program signals for a condition handler.

Source Program Number 2

```
PROGRAM Relative( INPUT, OUTPUT);

(*   RELATIVE.PAS                                       *)
(*   This program illustrates accessing a relative file  *)
(*   randomly. It also performs some I/O status checks.  *)

CONST  Prompt= 'Record number (CTRL/Z to quit): ';          ❶
       %INCLUDE 'SYS$LIBRARY:PASSTATUS.PAS'


TYPE Char_Array = PACKED ARRAY [1..20] OF CHAR;


VAR  Rel_File: FILE OF Char_Array;
     Rec_Num : INTEGER;


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

BEGIN   (* main program *)
   OPEN (Rel_File, 'REL.DAT', HISTORY:= OLD,
         ORGANIZATION:= RELATIVE, ACCESS_METHOD:= DIRECT);   ❷

   (* Get records by record number until EOF  *)
   WRITE (Prompt);
   WHILE NOT EOF (Input) DO
      BEGIN
      (* Get record *)
      READLN (Rec_Num);
      FIND (Rel_File, Rec_Num, ERROR:= CONTINUE);

      CASE STATUS (Rel_File) OF                              ❸
         PAS$K_SUCCESS: IF NOT UFB (Rel_File)
               THEN WRITELN (Rel_File^)
               ELSE WRITELN ('Did not find record ',Rec_Num);

         PAS$K_ERRDURFIN:  WRITELN ('Error during FIND.');

         OTHERWISE  (* signal the error *)
            LIB$STOP(%x218004 + ((STATUS(Rel_File) + 200) * 8))
```

```
        END;    (* end case *)
     WRITE (Prompt)
     END  (* while *)
END.
```

Sample Use

```
$ RUN RELATIVE
Record number (CTRL/Z to quit): 7
08001FLANJE119PL1920
Record number (CTRL/Z to quit): 1
07672ALBEHA210SE2100
Record number (CTRL/Z to quit): 30
Did not find record         30
Record number (CTRL/Z to quit): CTRL/Z
$
```

❶ Including PASSTATUS.PAS defines the VAX PASCAL status codes detected by the STATUS and STATUSV functions.

❷ This statement defines the file and record processing characteristics. Although a file organization of relative is specified, RMS would in fact obtain the file organization from an existing file. If the file organization were not relative, the file open statement would fail.

❸ The call to the FIND routine positions the file at the component specified in Rec_Num. Error := CONTINUE is specified, because the program will attempt to handle errors from the FIND procedure.

❹ The STATUS function is used to detect file access errors. The program performs certain operations if the status is SUCCESS or ERRDURFIN. If any other errors occur, LIB$STOP is called.

Source Program 3

```
PROGRAM Indexed (INPUT, OUTPUT);

(*   INDEXED.PAS                                       *)
(*   This program illustrates keyed and sequential     *)
(*   access to an indexed sequential file.  It prompts *)
(*   for a department number then prints the name of   *)
(*   each person in the department.                    *)

LABEL 100;

CONST Prompt = 'Enter department no. (119,210,220): ';
```

```
TYPE  Employee_Rec = RECORD
                     Id_Num: PACKED ARRAY [1..5] OF CHAR;
                     Name:   PACKED ARRAY [1..6] OF CHAR;
                     Dept:   PACKED ARRAY [1..3] OF CHAR;
                     Skill:  PACKED ARRAY [1..2] OF CHAR;
                     Salary: PACKED ARRAY [1..4] OF CHAR;
                     END;

VAR  Employees : FILE OF Employee_Rec;
     Dept_Key  : PACKED ARRAY [1..3] OF CHAR;

BEGIN
   (* Open indexed file *)
   OPEN (Employees, 'IDX.DAT', History:= OLD,                  ❶
         Organization:= INDEXED, Access_Method:= KEYED);

   (* Obtain department number *)
   WRITELN ('Type CTRL/Z to quit');
   WRITE (Prompt);
   WHILE NOT EOF (INPUT) DO
      BEGIN
         READLN (Dept_key);

         (* Read first dept_key record *)                      ❷
         FINDK (Employees, Key_Number:= 0,
                Key_Value:= Dept_Key);

         (* If valid dept., output members *)
         IF NOT UFB (Employees)
            THEN (* valid dept number *)
               WHILE NOT EOF (Employees) DO
                  BEGIN
                     IF Employees^.Dept <> Dept_Key
                     THEN GOTO 100;
                  WRITELN (Employees^.Name);                   ❸
                  GET (Employees)
                  END   (* output dept. members *)

            ELSE  (* not valid dept. number *)
               WRITELN ('Invalid department number.');
```

```
100:     (* Prompt for another dept. number *)
         WRITE (Prompt)
      END  (* not CTRL/Z *)
END.
```

Sample Use

```
$ RUN INDEXED
Type CTRL/Z to quit
Enter department no. (119,210,220): 119
ANDEWF
DALLJE
FLANJE
FLINGA
GREEJW
JONEKB
MANKCA
MARSJJ
REDFBB
SCHAWE
WIENSH
Enter department no. (119,210,220):  3
Invalid department number.
Enter department no. (119,210,220):  CTRL/Z
$
```

❶ To allow random keyed access to an indexed file, the access method parameter to the OPEN procedure must be INDEXED.

The location of the keys in the records need not be specified since this information is obtained by RMS from the file prologue.

❷ The FINDK procedure is used to locate a record by key value. A Key_Number of zero indicates the primary key, and Key_Value specifies the value of the key.

It is not necessary to call RESETK, because the file need not be in inspection mode before FINDK is executed.

❸ Sequential reads are performed to obtain the records with the appropriate department number. RMS will not return a different status code when the key value changes. Consequently, the program must check the key value in the record to determine whether or not it has changed.

Source Program 4

This example uses LIB$Find_File and LIB$Find_File_End to process a wildcarded filename expression.

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Find_File(INPUT,OUTPUT);

VAR
    File_Spec : VARYING [132] OF CHAR;
    Result_Spec : VARYING [132] OF CHAR;
    Context : UNSIGNED;
    Ret_Stat : UNSIGNED;

[EXTERNAL] FUNCTION LIB$Find_File(
        File_Spec    : VARYING [len1] OF CHAR;
    VAR Result_Spec  : VARYING [len2] OF CHAR;
    VAR Context      : UNSIGNED;
        Default_Spec: VARYING [len3] OF CHAR := %IMMED 0;
        Related_Spec: VARYING [len4] OF CHAR := %IMMED 0;
    VAR Stv_Addr     : UNSIGNED := %IMMED 0;
        User_Flags   : UNSIGNED := %IMMED 0
    ) : UNSIGNED; EXTERNAL;

[EXTERNAL] FUNCTION LIB$Find_File_End(
        Context      : UNSIGNED
    ) : UNSIGNED; EXTERNAL;

[EXTERNAL] FUNCTION LIB$Stop(
        Condition : [IMMEDIATE] UNSIGNED;
        Fao_Args  : [IMMEDIATE,LIST,UNSAFE] UNSIGNED
        ) : UNSIGNED; EXTERNAL;

BEGIN
Context := 0;

WRITE('Enter filespec to parse: ');
WHILE NOT EOF DO
        BEGIN

        { Read the filespec from the user }
        READLN(File_Spec);

        { Loop and parse the file spec }
        REPEAT

            { Parse it... }
            Ret_Stat := LIB$Find_File(           ❶
                File_Spec,
                Result_Spec,
                Context);
            IF (NOT ODD(Ret_Stat)) AND        ❷
               (Ret_Stat <> RMS$_NMF) AND
               (Ret_Stat <> RMS$_FNF)
            THEN
                LIB$Stop(Ret_Stat);

            IF (Ret_Stat <> RMS$_NMF) AND
               (Ret_Stat <> RMS$_FNF)
            THEN
                WRITELN(Result_Spec);

        UNTIL (Ret_Stat = RMS$_NMF) OR (Ret_Stat = RMS$_FNF);  ❸

        { Clear LIB$FIND_FILE Context }
        LIB$Find_File_End(Context);        ❹

        { Get another file spec }
        WRITE('Enter filespec to parse: ');

        END;
END.
```

❶ Call LIB$FIND_FILE to return a filename that satifies the wildcard file specification.

❷ Determine if LIB$FIND_FILE successfully returned a filename. Termination because of no more files (RMS$_NMF) or no such file (RMS$_FNF) will be ignored.

❸ Continue calling LIB$FIND_FILE until there are no more files that satisfy the wildcard file specification.

❹ Call LIB$FIND_FILE_END to deallocate the context from the calls to LIB$FIND_FILE so they won't affect the program when it parses for the next wildcard file specification.

Source Program 5

```
[INHERIT('SYS$LIBRARY:STARLET')]

MODULE Pas_Name;

TYPE
    Unsigned_Word   = [WORD] 0..65535;
    Ptr_To_Fab      = ^FAB$TYPE;
    Ptr_To_Nam      = ^NAM$TYPE;
    Unsafe_File     = [UNSAFE] FILE OF CHAR;
    File_Name_Type  = VARYING [NAM$C_MAXRSS] OF CHAR;


[EXTERNAL] FUNCTION PAS$Fab
    (VAR F : Unsafe_File) : Ptr_To_Fab; EXTERNAL;
```

```
[EXTERNAL] FUNCTION Str$Copy_R
    (VAR Dst_Str     : VARYING [u1] OF CHAR;
         Src_Len     : Unsigned_Word;
         Src_Str     : [IMMEDIATE] UNSIGNED) : INTEGER; EXTERNAL;


[GLOBAL] FUNCTION Pas_Name
    (VAR F : Unsafe_File) : File_Name_Type;

    {+}
    {
    {       Return the file-name of external file associated
    {       with a file-variable.
    {
    {       This function assumes that the calling routine
    {       has declared this function as returning a
    {       VARYING [NAM$C_MAXRSS] OF CHAR.  If declared
    {       as VARYING with a size less than 255 the
    {       results will be unpredictable.
    {
    {-}

    VAR
        Name     : File_Name_Type;       { temp holder for name }
        Fab      : Ptr_To_Fab;           { RMS FAB }
        Nam      : Ptr_To_Nam;           { RMS NAM }
        Len      : UNSIGNED_WORD;
        Ptr      : UNSIGNED;

    BEGIN
    Name.LENGTH := 0;

    { get RMS FAB for file variable }

    Fab := PAS$Fab( F );
    IF Fab <> NIL
    THEN
        BEGIN
        Nam := Fab^.FAB$l_Nam :: Ptr_To_Nam;
        IF Nam = NIL
        THEN
            BEGIN
            { copy file-name field from FAB }
            Len := Fab^.FAB$b_Fns;
            Ptr := Fab^.FAB$l_Fna;
            END
        ELSE
            BEGIN
            { copy resultant-name from NAM }
            Len := Nam^.NAM$b_Rsl;
            Ptr := Nam^.NAM$l_Rsa;
            END;

        Str$Copy_R( Name, Len, Ptr );
        END;
    Pas_Name := Name;
    END;

END.
```

E.8 Measuring and Improving Performance

This example illustrates how to adjust the size of the process working set from a program.

Source Program

```
[INHERIT ('SYS$LIBRARY:STARLET.PEN')] PROGRAM Adjust( OUTPUT);

(*   ADJUST.PAS                                             *)
(*   This program illustrates how a program can control its *)
(*   working set size using the $ADJWSL system service.     *)

CONST   Index = 10;

VAR     Adjust_Amt:     INTEGER;
        New_Limit:      INTEGER;
        Sys_Stat:       INTEGER;

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

BEGIN
   Adjust_Amt:=-50;
   WHILE Adjust_Amt <= 70 DO
      BEGIN
         (* Modify working set limit *)
         Sys_Stat:= $ADJWSL(Adjust_Amt,New_Limit);           ❶
         IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);

         WRITELN ('Modify working set by ',Adjust_Amt,
            '  New working set size = ',New_Limit);
         Adjust_Amt:= Adjust_Amt + 10
      END
END.
```

Sample Use

```
$ SHOW WORKING_SET                                           ❷
  Working Set     /Limit=  150    /Quota=  200            /Extent=  200
  Adjustment enabled   Authorized Quota=  200  Authorized Extent=  300
```

```
$ RUN ADJUST
Modify working set by -50    New working set size = 100
Modify working set by -40    New working set size = 60
Modify working set by -30    New working set size = 40
Modify working set by -20    New working set size = 40         ❸
Modify working set by -10    New working set size = 40
Modify working set by   0    New working set size = 40
Modify working set by  10    New working set size = 50
Modify working set by  20    New working set size = 70
Modify working set by  30    New working set size = 100
Modify working set by  40    New working set size = 140
Modify working set by  50    New working set size = 190
Modify working set by  60    New working set size = 200        ❹
Modify working set by  70    New working set size = 200
$
```

❶ The $ADJWSL is used to increase or decrease the number of pages in the process working set.

❷ The SHOW WORKING_SET command in DCL displays the current working set limit and the maximum quota.

❸ Notice that the program cannot decrease the working set limit beneath the minimum established by the operating system.

❹ Similarly, the process working set cannot be expanded beyond the authorized quota.

E.9 Accessing Help Libraries

The following example illustrates how to obtain text from a help library. After the initial help request has been satisfied, the user is prompted and can request additional information.

Source Program

```
PROGRAM Gethelp (INPUT, OUTPUT);

(*      GETHELP.PAS                                     *)
(*      This program will look up help text in          *)
(*      SYS$HELPLIB.HLB and displays it.                *)

CONST LBR$C_READ = %X'01';                             ❶


TYPE Word_Integer = [WORD] 1..65535;
     Char_String = VARYING [32] OF CHAR;
     Int_Array = ARRAY [1..3] OF INTEGER;


VAR  Key     : ARRAY [1..3] OF Char_String;
     Key_Len : ARRAY [1..3] OF INTEGER;
     Lib_Index, Help_Stat, Counter: INTEGER;
```

```
(* Declare external RTL routines *)

FUNCTION LBR$INI_CONTROL (VAR Library_Index : INTEGER;
                          Func : INTEGER;
                          Libe_Type : INTEGER := %IMMED 0;
                          VAR Namblk: ARRAY [1..U:INTEGER]
                                OF INTEGER := %IMMED 0):
                          INTEGER; EXTERN;


FUNCTION LBR$OPEN (Library_Index : INTEGER;
                   Fns : [CLASS_S] PACKED ARRAY [1..U:INTEGER]
                         OF CHAR := %IMMED 0;
                   Create_Options: Int_Array := %IMMED 0;
                   Dns : [CLASS_S] PACKED ARRAY [12..u2:INTEGER]
                         OF CHAR := %IMMED 0;
                   Rlfna : ARRAY [13..u3:INTEGER] OF INTEGER
                         := %IMMED 0;
                   Rns : [CLASS_S] PACKED ARRAY [14..u4:INTEGER]
                         OF CHAR := %IMMED 0;
                   VAR Rnslen : INTEGER := %IMMED 0): INTEGER;
                                EXTERN;


FUNCTION LBR$GET_HELP (Library_Index : INTEGER;
                       Line_Width : INTEGER := %IMMED 0;
                       %IMMED [UNBOUND] PROCEDURE Routine
                              := %IMMED 0;
                       data : INTEGER := %IMMED 0;
                       Key_1 : [CLASS_S] PACKED ARRAY [1..U:
                               INTEGER] OF CHAR;
                       Key_2 : [CLASS_S] PACKED ARRAY [12..u2:
                               INTEGER] OF CHAR;
                       Key_3 : [CLASS_S] PACKED ARRAY [13..u3:
                               INTEGER] OF CHAR): INTEGER;
                               EXTERN;

FUNCTION LBR$CLOSE (Library_Index : INTEGER): INTEGER; EXTERN;

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

BEGIN           (* main program *)

   (* Initialize the librarian *)
   Help_Stat := LBR$INI_CONTROL (Lib_Index, LBR$C_READ,,);        ❷
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);

   (* Open the library *)
   Help_Stat := LBR$OPEN (Library_Index := Lib_Index,             ❸
                          Fns := 'SYS$HELP:HELPLIB.HLB');
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);
   (* Get HELP keys *)
   FOR Counter := 1 TO 3 DO
      BEGIN
      WRITE ('Enter key: ');
      READLN (Key[Counter]);                               ❹
      END;

   (* Locate and print the help text *)                    ❺
   Help_Stat:=LBR$GET_HELP (Lib_Index,,,,Key[1], Key[2], Key[3]);
   IF NOT ODD (Help_Stat) THEN LIB$STOP (Help_Stat);
END.
```

Sample Use

```
$ RUN GETHELP
Enter Key : REQUEST
Enter Key : /REPLY
Enter Key :

  REQUEST

     /REPLY

      Requests a reply to the specified message.

      If you request a  reply, the  message is  assigned a unique
      identification so that the operator can respond.  You will
      not be able to continue until the operator responds, unless
      you use CTRL/Y.
$
```

❶ The LBR$C_READ symbolic code is defined as a constant (with a hexadecimal value). The value associated with this particular code is obtained from the $LBRDEF macro in STARLET.MLB.

The code is used to specify the operation that is to be performed on a library.

❷ The call to LBR$INI_CONTROL initializes the library index and defines the operation to be performed on the library.

❸ The call to LBR$OPEN identifies the library to be accessed, in this case the system help library, SYS$HELP:HELPLIB.HLB.

❹ The user is asked to supply the keys to look up help text. In this case Key[1] corresponds to a DCL command, while Key[2] and Key[3] (optional) are command parameters or qualifiers.

❺ If no routine name is specified in the call to LBR$GET_HELP, the returned help text is written to SYS$OUTPUT (with line-width defaulting to 80). Note that several special symbols can be used for the key arguments:

*	All first-level help text in the library.
KEY...	All help text with specified key and its subkeys.
*...	All help text in the library.

E.10 Creating and Managing Other Processes

The following example illustrates the ability of a created process to use the SYS$GETJPIW system service to obtain the PID of its creator process.

Source Program 1

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Getjpi( OUTPUT);

(*   GETJPI.PAS                                              *)
(*   This program illustrates process creation and control.  *)
(*   It creates a subprocess then hibernates until the       *)
(*   subprocess wakes it.                                    *)

CONST   File_Name = 'GETJPISUB';
        Sub_Name = 'OSCAR';
        Esc_Null = ''(27)''(0)'';       (* ESCAPE/NULL string *)

TYPE  $UBYTE = [BYTE] 0..255;
        Word_Integer = [WORD] 0..65535;
        Dummy_Rec = RECORD
                    END;
        Item_List = RECORD
                    Buf_Len1      : Word_Integer;
                    Item_Code1    : Word_Integer;
                    Buffer_Addr1  : [UNSAFE] ^[UNSAFE]Dummy_Rec;
                    Ret_Len_Addr1: ^Word_Integer;
                    Buf_Len2      : Word_Integer;
                    Item_Code2    : Word_Integer;
                    Buffer_Addr2  : ^INTEGER;
                    Ret_Len_Addr2: ^Word_Integer;
                    Terminator    : INTEGER
                    END;
```

```
VAR     Terminal  :    [VOLATILE] VARYING [255] OF CHAR;
        Equiv_Name:    [VOLATILE] PACKED ARRAY [1..255] OF CHAR;
        Process_Id:    UNSIGNED;
        New_Term_Len: Word_Integer;
        Equiv_Name_Len,Info_Len : [VOLATILE] Word_Integer;
        Tran_Mask :    [VOLATILE] INTEGER;
        Sys_Stat  :    INTEGER;
        Not_Terminal: BOOLEAN := TRUE;
        Trn_List  :    Item_List;
        Unsigned_Mask,Term_Attr : UNSIGNED;


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

BEGIN
   (* Initialize *)
   Term_Attr:=LNM$M_TERMINAL;
   Process_Id:= 0;
   Terminal:= 'SYS$OUTPUT';
   Trn_List.Buf_Len1       := 255;
   Trn_List.Item_Code1     := LNM$_STRING;
   Trn_List.Buffer_Addr1   := ADDRESS (Equiv_Name);
   Trn_List.Ret_Len_Addr1 := ADDRESS (Equiv_Name_Len);
   Trn_List.Buf_Len2       := 4;
   Trn_List.Item_Code2     := LNM$_ATTRIBUTES;
   Trn_List.Buffer_Addr2   := ADDRESS (Tran_Mask);
   Trn_List.Ret_Len_Addr2 := ADDRESS (Info_Len);
   Trn_List.Terminator     := 0;

   (* Translate SYS$OUTPUT *)
   WHILE Not_Terminal AND (Sys_Stat <> SS$_NOLOGNAM) DO
       BEGIN
       Sys_Stat:= $TRNLNM (Tabnam := 'LNM$PROCESS',
                           Lognam := Terminal,
                           Itmlst := Trn_List);
       Terminal := SUBSTR (Equiv_Name,1,Equiv_Name_Len);
       Unsigned_Mask := Tran_Mask;
       IF UAND (Unsigned_Mask,Term_Attr) <> 0
           THEN Not_Terminal := FALSE;
       END;

   (* Check if process permanent file *)
   IF (SUBSTR (Terminal,1,2) = Esc_Null)
      THEN
         BEGIN
         (* Strip off extra 4 bytes *)
         New_Term_Len:= Equiv_Name_Len - 4;
         Terminal:= SUBSTR (Terminal,5,New_Term_Len);
         END;
   (* Create the subprocess *)
   Sys_Stat:=$CREPRC (Pidadr := Process_Id, Image:=File_Name, ❶
                      Input  := Terminal, Output:= Terminal,
                      Prcnam := Sub_Name, Baspri:= 4);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('PID of subprocess OSCAR is ',HEX(Process_Id));

   (* Wait for wakeup by subprocess *)
   Sys_Stat:= $HIBER;
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   WRITELN ('GETJPI has been awakened.')
END.
```

Source Program 2

```
[INHERIT ('SYS$LIBRARY:STARLET')] PROGRAM Getjpisub( OUTPUT);

(*   GETJPISUB.PAS                                           *)
(*   This program is run in the subprocess OSCAR which is   *)
(*   created by GETJPI.  It obtains its creator's PID then  *)
(*   wakes it.                                              *)

TYPE    Word_Integer = [WORD] 0..65535;
        Getjpi_List = RECORD
                        Buf_Len      : Word_Integer;
                        Item_Code    : Word_Integer;
                        Buffer_Addr  : ^UNSIGNED;
                        Ret_Len_Addr: ^INTEGER;
                        Must_Be_Zero: INTEGER;
                        END;

        Io_Block = RECORD
                     Io_Stat, Count: Word_Integer;
                     Dev_Info : INTEGER;
                     END;

VAR     Jpi_List: Getjpi_List;
        Stat_Block: Io_Block;
        Sys_Stat: INTEGER;
        Buf_Val : [VOLATILE] UNSIGNED;
        Ret_VAl : [VOLATILE] INTEGER;

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

BEGIN
   (* Initialize *)
   Jpi_List.Buf_Len      := 4;
   Jpi_List.Item_Code    := JPI$_OWNER;
   Jpi_List.Buffer_Addr := ADDRESS (Buf_Val);                ❷
   Jpi_List.Ret_Len_Addr:= ADDRESS (Ret_Val);
   Jpi_List.Must_Be_Zero:= 0;
   (* Get PID of creator *)
   Sys_Stat:= $GETJPI (Efn:= 1,Iosb:=Stat_Block,Itmlst:=
                       Jpi_List);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);          ❸
   Sys_Stat:= $SYNCH (Efn:=1,Iosb:=Stat_Block);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat);
   IF NOT ODD (Stat_Block.Io_Stat) THEN
               LIB$STOP (Stat_Block.Io_Stat);
   (* Wake creator *)
   WRITELN ('   OSCAR is waking creator.');                  ❹
   Sys_Stat:= $WAKE (Jpi_List.Buffer_Addr^,);
   IF NOT ODD (Sys_Stat) THEN LIB$STOP (Sys_Stat)
END.
```

Sample Use

```
$ RUN GETJPI
PID of subprocess OSCAR is 0002002E
   OSCAR is waking creator.
GETJPI has been awakened.
$
```

❶ The subprocess is created using SYS$CREPRC.

❷ The item list for $GETJPI consists of a single item descriptor followed by a zero longword. JPI$_OWNER is the item code which requests the PID of the owner process. If there is no owner process (i.e., if the process about which information is requested is a detached process), the system service $GETJPI returns a PID of zero.

❸ Because the item code JPI$_OWNER is in the item list, $GETJPI returns the PID of the owner of the process about which information is requested. If the item code were JPI$_PID, $GETJPI would return the PID of the process about which information is requested.

Because the default value of 0 is used for arguments PIDADR and PRCNAM, the process about which information is requested is the requesting process, namely, OSCAR.

❹ OSCAR wakes its owner process using the PID it received from the call to $GETJPI.

E.11 Translating Logical Names

The following example illustrates the common task of translating logical names.

Source Program

```
[INHERIT('SYS$LIBRARY:STARLET')]
PROGRAM Translate(INPUT,OUTPUT);

VAR
   Input_String  : VARYING [132] OF CHAR;
   Output_String : VARYING [132] OF CHAR;
   Return_Status : INTEGER;
   Trnlnm_Item_List : RECORD
                      Buffer_Length  : [WORD] 0..65535;
                      Item_Code      : [WORD] 0..65535;
                      Buffer_Address: INTEGER;
                      Return_Length  : INTEGER;
                      End_Of_List    : INTEGER;
                      END;


BEGIN

(* Prompt user for logical name to translate *)
WRITE('Logical name? ');
READLN(Input_String);

(* Build item list *)
WITH Trnlnm_Item_List DO
    BEGIN
    Buffer_Length  := SIZE(Output_String.Body);
    Item_Code      := LNM$_STRING;
    Buffer_Address:= IADDRESS(Output_String.Body);          ❶
    Return_Length  := IADDRESS(Output_String.Length);
    End_Of_List    := 0;
    END;


(* Call TRNLNM to translate logical name *)
RETURN_STATUS := $TRNLNM(
      ATTR    := LNM$M_CASE_BLIND,
      TABNAM  := 'LNM$SYSTEM_TABLE',
      LOGNAM  := Input_String,
      ITMLST  := Trnlnm_Item_List);


IF RETURN_STATUS = SS$_NORMAL
THEN
      WRITELN('Translated name is ',Output_String)

ELSE IF RETURN_STATUS = SS$_NOLOGNAM
THEN
      WRITELN('No translation for logical name ',Input_String)
```

```
ELSE
      WRITELN('Error in TRNLNM routine');

END.
```

Sample Use

```
$ RUN TRNLNM
Logical name?  sys$batch
Translated name is  CLU_BATCH

$ RUN TRNLNM
Logical name?  hello
No translation for logical name hello
```

❶ Build the item list to place the result directly into a VARYING OF CHAR variable. This is done by computing the address of the length word and body.

Appendix F

Optional Programming Productivity Tools

This appendix provides an overview of optional programming productivity tools. These tools are not included with the VAX PASCAL software; they must be purchased separately. Using these tools can increase your productivity as a VAX PASCAL programmer.

F.1 Using VAXLSE with VAX PASCAL

The VAX Language-Sensitive Editor (VAXLSE) is a powerful and flexible text editor designed specifically for software development. VAXLSE has important features that help you produce syntatically correct code in VAX PASCAL.

To invoke VAXLSE, specify the LSEDIT command followed by a file name with a PAS file type at the DCL prompt. For example:

```
$ LSEDIT USER.PAS
```

The following sections describe some of the key features of VAXLSE. Section F.1.1 discusses how to enter source code using VAXLSE and Section F.1.2 describes VAXLSE's compiler interface features. Section F.1.3 gives examples of how to generate VAX PASCAL source code with VAXLSE.

For more details on advanced features of VAXLSE and VAXSCA, see the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer.*

F.1.1 Entering Source Code Using Tokens and Placeholders

VAXLSE's language-sensitive features simplify the tasks of developing and maintaining software systems. These features include language-specific placeholders and tokens, aliases, comment and indentation control, and templates for subroutine libraries.

VAXLSE can be used as a traditional text editor. In addition, you can have the power of using VAXLSE's tokens and placeholders to step through each program construct and supply text for those constructs needing it.

Placeholders are markers in the source code that indicate locations where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. Generally, you do not need to type placeholders; rather, they are inserted for you by VAXLSE. Placeholders are surrounded by brackets or braces and percent signs.

The types of VAXLSE placeholders are as follows:

Type	Description
Terminal placeholders	Provide text strings that describe valid replacements for the placeholder
Nonterminal placeholders	Expand into additional language constructs
Menu placeholders	Provide a list of options corresponding to the placeholder

Placeholders are either optional or required. Required placeholders, indicated by braces ({}), represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets ([]), represent places in the source code where you can either provide additional constructs or erase the placeholder.

You can move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed.

Tokens typically represent keywords in VAX PASCAL. When expanded, tokens provide additional language constructs. You can type tokens directly into the buffer. Generally, you use tokens in situations, such as modifying an existing program, where you want to add additional language constructs and there are no placeholders. For example, typing IF and issuing the EXPAND command causes a template for an IF construct to appear on your screen. You can also use tokens to bypass long menus in situations where expanding a placeholder, such as %{statement}%, would result in a lengthy menu.

You can use tokens to insert text when editing an existing file by typing the name for a function or keyword and issuing the EXPAND command.

VAXLSE provides commands that allow you to manipulate tokens and placeholders. These commands and their default key bindings are as follows:

Command	**Key Binding**	**Function**
EXPAND	CTRL/E	Expands a placeholder
UNEXPAND	PF1-CTRL/E	Reverses the effect of the most recent placeholder expansion
GOTO PLACEHOLDER /FORWARD	CTRL/N	Moves the cursor forward to the next placeholder
GOTO PLACEHOLDER /REVERSE	CTRL/P	Moves the cursor backward to the previous placeholder
ERASE PLACEHOLDER /FORWARD	CTRL/K	Erases a placeholder
UNERASE PLACEHOLDER	PF1-CTRL/K	Restores the most recently erased placeholder
	Down arrow	Moves the indicator through a screen menu toward the bottom
	Up arrow	Moves the indicator through a screen menu toward the top
	$\left\{\begin{matrix}\text{ENTER}\\\text{RETURN}\end{matrix}\right\}$	Selects a menu option

To display a list of all the defined tokens provided by VAX PASCAL, enter the SHOW TOKEN command:

```
LSE> SHOW TOKEN
```

To display a list of all the defined placeholders provided by VAX PASCAL, enter the SHOW PLACEHOLDER command:

```
LSE> SHOW PLACEHOLDER
```

To put a copy of either list into a separate file, first enter the appropriate SHOW command to put the list into the $SHOW buffer. Then enter the following commands:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename
```

To obtain a hard copy of the list, use the PRINT command at DCL level to print the file you created.

To obtain information about a particular token or placeholder, you can also specify a token name or placeholder name after the SHOW TOKEN or SHOW PLACEHOLDER command.

F.1.2 Compiling Source Code

To compile your code and to review compilation errors without leaving the editing session, you can use the VAXLSE commands COMPILE and REVIEW. The COMPILE command issues a DCL command in a subprocess to invoke the VAX PASCAL compiler. The compiler then generates a file of compile-time diagnostic information that VAXLSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that VAXLSE appends onto the compilation command.

For example, if you issue the COMPILE command while in the buffer USER.PAS, the resulting DCL command is as follows:

```
$ PASCAL USER.PAS/DIAGNOSTICS=USER.DIA
```

VAXLSE supports all of the VAX PASCAL compiler's command qualifiers as well as user-supplied command procedures. You can specify DCL qualifiers, such as the /LIBRARY qualifier, when invoking the compiler from VAXLSE.

The REVIEW command displays any diagnostic messages that result from a compilation. VAXLSE displays the compilation errors in one window and the corresponding source code in a second window. This multiwindow capability allows you to review your errors while examining the associated source code. This capability eliminates tedious steps in the error correction process, and helps ensure that all the errors are fixed before you compile your program again.

VAXLSE provides several commands to help you review errors and examine your source code. The following table lists these commands and their default key bindings where applicable.

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer
COMPILE/REVIEW	None	Compiles the contents of the source buffer, puts VAXLSE into REVIEW mode, and displays any errors resulting from the compilation
REVIEW	None	Performs the same function as the /REVIEW qualifier on the COMPILE command: puts VAXLSE into REVIEW mode, and displays any errors resulting from the last compilation
END REVIEW	None	Removes the buffer $REVIEW from the screen and returns the cursor to a single window containing the source buffer
GOTO SOURCE	CTRL/G	Moves the cursor to the source buffer that contains the error
NEXT STEP	CTRL/F	Moves the cursor to the next error in the buffer $REVIEW
PREVIOUS STEP	CTRL/B	Moves the cursor to the previous error in the buffer $REVIEW
	{ Down arrow / Up arrow }	Moves the cursor within a buffer

F.1.3 Examples

This section describes the special features of VAX PASCAL available through VAXLSE and provides examples of VAX PASCAL code written with VAXLSE.

The following examples show expansions of the more frequently used VAX PASCAL tokens and placeholders. The examples are expanded to show the formats and guidelines VAXLSE provides; however, not all of the examples are fully expanded.

The examples show expansions of the following VAX PASCAL features:

- TYPE definition

- FOR statement
- OPEN statement

Instructions and explanations precede each example, and an arrow (←) indicates the line in the code where an action has occurred.

To reproduce the examples, invoke VAXLSE and the VAX PASCAL language by using the following syntax:

```
   LSEDIT [/qualifier...] filename.PAS
```

See Section F.1.1 for the commands that manipulate tokens and placeholders.

Remember that braces and percent signs (%{}%) enclose required placeholders; brackets and percent signs (%[ ]%) enclose optional placeholders. Note that when you erase an optional placeholder, VAXLSE also deletes any associated text before and after that placeholder.

NOTE

> Keywords such as TYPE, VAR, IF, FOR, OPEN, and WRITELN can be tokens as well as placeholders; therefore, any time you are in the VAX PASCAL language environment you can type one of these words and press CTRL/E to cause the expansion to appear.

When you use VAXLSE to create a new VAX PASCAL program, the initial string, %{compilation_unit}%, appears at the top of the screen. Expand the initial string to produce a menu, and select the option PROGRAM. The following then appears on your screen:

```
[%[comp_unit_attribute]%...]   <--
PROGRAM %{user_defined_id}% (%[program_parameter]%...);

%[header_comments]%

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Erase the placeholder %[comp_unit_attribute]% and type EXAMPLE over the placeholder %{user_defined_id}%.

Erase the placeholders %[program_parameter]% and %[header_comments]%.

```
PROGRAM EXAMPLE;  <--

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

F.1.3.1 TYPE Definition

Beginning where you were after erasing the placeholder %[header_comments]%, move to the placeholder %[declaration_list]%, type TYPE over it, and expand TYPE.

```
PROGRAM EXAMPLE;

TYPE                          <--
    %{type_definition}%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Expand the placeholder %{type_definition}% to display a menu, and select the option %{type_list}%.

```
PROGRAM EXAMPLE;

TYPE
    %{type_id}% = [%[variable_attributes]%]... %{data_type}%; <--
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Type REC over the placeholder %{type_id}% and erase the placeholder %[variable_attributes]%.

```
PROGRAM EXAMPLE;

TYPE
    REC = %{data_type}%;      <--
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Type RECORD over the placeholder %{data_type}% and expand RECORD.

```
PROGRAM EXAMPLE;

TYPE
    REC = %[PACKED]% RECORD    <--
              %[field_list]%
          END;
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Erase the placeholder %[PACKED]% and expand the placeholder %[field_list]%.

```
PROGRAM EXAMPLE;

TYPE
    REC = RECORD                         <--
              %[field_components]%...    <--
              %[variant_clause]%
          END;
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Expand the placeholder %[field_components]%.

```
PROGRAM EXAMPLE;

TYPE
    REC = RECORD
              %{field_id}%... : [%[component_attributes]%...] %{data_type}%;  <--
              %[field_components]%...
              %[variant_clause]%
          END;
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Type FIELD1 over the list placeholder %{field_id}%. Erase the duplicated placeholder %[field_id]% and the placeholder %[component_attributes]%. Type CHAR over the placeholder %{data_type}%.

```
PROGRAM EXAMPLE;

TYPE
    REC = RECORD
              FIELD1 : CHAR;          <--
              %[field_components]%...
              %[variant_clause]%
          END;
    %[type_definition]%...;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

Type FIELD2 : REAL over the list placeholder %[field_components]%. Erase the duplicated placeholder %[field_components]%, the placeholder %[variant_clause]%, and the placeholder %[type_definition]%.

```
PROGRAM EXAMPLE;

TYPE
    REC = RECORD
              FIELD1 : CHAR;
              FIELD2 : REAL
          END;

%[declaration_list]%...

BEGIN
%[statement_list]%...
END.
```

F.1.3.2 FOR Statement

Move your cursor to the statement block shown at the end of the previous example.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
%[statement_list]%...
END.
```

Type FOR over the placeholder %[statement_list]% and expand FOR.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
FOR %{control_var}% := %{value_expr}% %{TO | DOWNTO}% %{value_expr}% DO  <--
    %{statement}%;
%[statement_list]%...
END.
```

Type INDEX over the placeholder %{control_var}% and the value 1 over the placeholder %{value-expr}%.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
FOR INDEX := 1 %{TO | DOWNTO}% %{value_expr}% DO   <--
    %{statement}%;
%[statement_list]%...
END.
```

Type TO over the placeholder %{TO | DOWNTO}% and MAX over the placeholder %{value_expr}%.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
FOR INDEX := 1 TO MAX DO    <--
    %{statement}%;
%[statement_list]%...
END.
```

Expand the placeholder %{statement}% to display a menu. Select the option %{simple_statement}% to display another menu and select the ASSIGNMENT option.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
FOR INDEX := 1 TO MAX DO
    %{variable | func_id}% := %{value_expr}%;   <--
%[statement_list]%...
END.
```

Type ARR[INDEX] over the placeholder %{variable | func_id}% and the value 0 over the placeholder %{value_expr}%.

```
PROGRAM EXAMPLE;
    .
    .
    .
BEGIN
FOR INDEX := 1 TO MAX DO
    ARR[INDEX] := 0;        <--
%[statement_list]%...
END.
```

F.1.3.3 OPEN Statement

Begin with your cursor in the statement block, as at the start of the previous example. To distinguish this exercise from the previous one, the comment (JOURNAL_ACCOUNTS) has been added to the program name.

```
PROGRAM EXAMPLE (JOURNAL_ACCOUNTS);
    .
    .
    .
BEGIN
%[statement_list]%...
END.
```

Type OPEN over the placeholder %[statement_list]% and expand OPEN to display a menu. Select the option %[nonpositional_open]%.

```
PROGRAM EXAMPLE (JOURNAL_ACCOUNTS);
    .
    .
    .
BEGIN
OPEN( FILE_VARIABLE     := %{file_variable}%,     <--
      %[file_name       :=   file_name]%,
      %[history         :=   file_history]%,
      %[record_length   :=   record_length]%,
      %[access_method   :=   access_method]%,
      %[record_type     :=   record_type]%,
      %[carriage_control:=   carriage_control]%,
      %[organization    :=   organization]%,
      %[disposition     :=   disposition]%,
      %[sharing         :=   file_sharing]%,
      %[user_action     :=   user_action]%,
      %[default         :=   filespec]%,
      %[error           :=   error_recovery]% );
%[statement_list]%...
END.
```

Type JOURNAL_ACCOUNTS over the placeholder %{file_variable}%.

```
PROGRAM EXAMPLE (JOURNAL_ACCOUNTS);
    .
    .
    .
BEGIN
OPEN( FILE_VARIABLE     := JOURNAL_ACCOUNTS,    <--
      %[file_name       :=  file_name]%,
      %[history         :=  file_history]%,
      %[record_length   :=  record_length]%,
      %[access_method   :=  access_method]%,
      %[record_type     :=  record_type]%,
      %[carriage_control:=  carriage_control]%,
      %[organization    :=  organization]%,
      %[disposition     :=  disposition]%,
      %[sharing         :=  file_sharing]%,
      %[user_action     :=  user_action]%,
      %[default         :=  filespec]%,
      %[error           :=  error_recovery]% );
%[statement_list]%...
END.
```

Expand the placeholder %[file_name := file_name]% and type JOURNAL.DAT over the placeholder file_name.

```
PROGRAM EXAMPLE (JOURNAL_ACCOUNTS);
    .
    .
    .
BEGIN
OPEN( FILE_VARIABLE     := JOURNAL_ACCOUNTS,
      FILE_NAME         := 'JOURNAL.DAT',      <--
      %[history         :=  file_history]%,
      %[record_length   :=  record_length]%,
      %[access_method   :=  access_method]%,
      %[record_type     :=  record_type]%,
      %[carriage_control:=  carriage_control]%,
      %[organization    :=  organization]%,
      %[disposition     :=  disposition]%,
      %[sharing         :=  file_sharing]%,
      %[user_action     :=  user_action]%,
      %[default         :=  filespec]%,
      %[error           :=  error_recovery]% );
%[statement_list]%...
END.
```

Expand the placeholder %[history := file_history]%, type UNKNOWN over the placeholder file_history, and erase the remaining placeholders.

```
PROGRAM EXAMPLE (JOURNAL_ACCOUNTS);
    .
    .
    .
BEGIN
OPEN( FILE_VARIABLE     := JOURNAL_ACCOUNTS,
      FILE_NAME         := 'JOURNAL.DAT',
      HISTORY           := UNKNOWN );       <--
%[statement_list]%...
END.
```

F.2 Using the VAX Source Code Analyzer

The VAX Source Code Analyzer (VAXSCA) is an interactive source code cross-reference and static analysis tool that works with most VMS programming languages. VAXSCA helps developers keep track of the details of complex, large-scale software systems by displaying source information in response to user queries. VAXSCA uses data generated by the VAX PASCAL compiler to supply the requested source information. That information is stored in a unique location, the VAXSCA library. The data in a VAXSCA library consists of the names of, and information about, all the symbols, modules, and files encountered during a specific compilation of the source.

VAXSCA has both *cross-reference* and *static analysis* query features. Cross-referencing supplies information about program symbols and source files. Cross-referencing features include the following:

- Locating names, and occurrences (uses) of these names
- Querying a specified set of names or partial names (wildcards allowed)
- Limiting a query to specific characteristics (such as routine names, variable names, or source files)
- Limiting a query to specific occurrences (such as the primary declaration of a symbol, read or write occurrences of a symbol, or occurrences of a file)

The static analysis query features of VAXSCA provide structural information on the interrelation of routines, symbols and files. Static analysis features include the following:

- Displaying routine calls to and from a specified routine
- Analyzing routine calls for consistency as to the numbers and data types of arguments passed, and the types of values returned

VAXSCA is fully integrated with VAXLSE to provide extended features. By using VAXSCA with VAXLSE, you can view any portion of an entire system and edit related source files.

Multimodular Development

The cross-referencing and static analysis features of VAXSCA can become useful during the implementation and maintenance phases of a project that involves many programming modules. For example, Figure F-1 shows a project team work area that contains a set of source modules. To keep track of these modules in their various development stages, the team can use a code management tool, such as VAX DEC/CMS, which is represented in the figure by the CMS library.

When the team compiles the source code, a /ANALYSIS_DATA qualifier to the COMPILE command instructs the VAX PASCAL compiler to generate VAXSCA-required source information (.ANA data files) from the sources. The team then instructs VAXSCA to load the .ANA files into a previously established VAXSCA library.

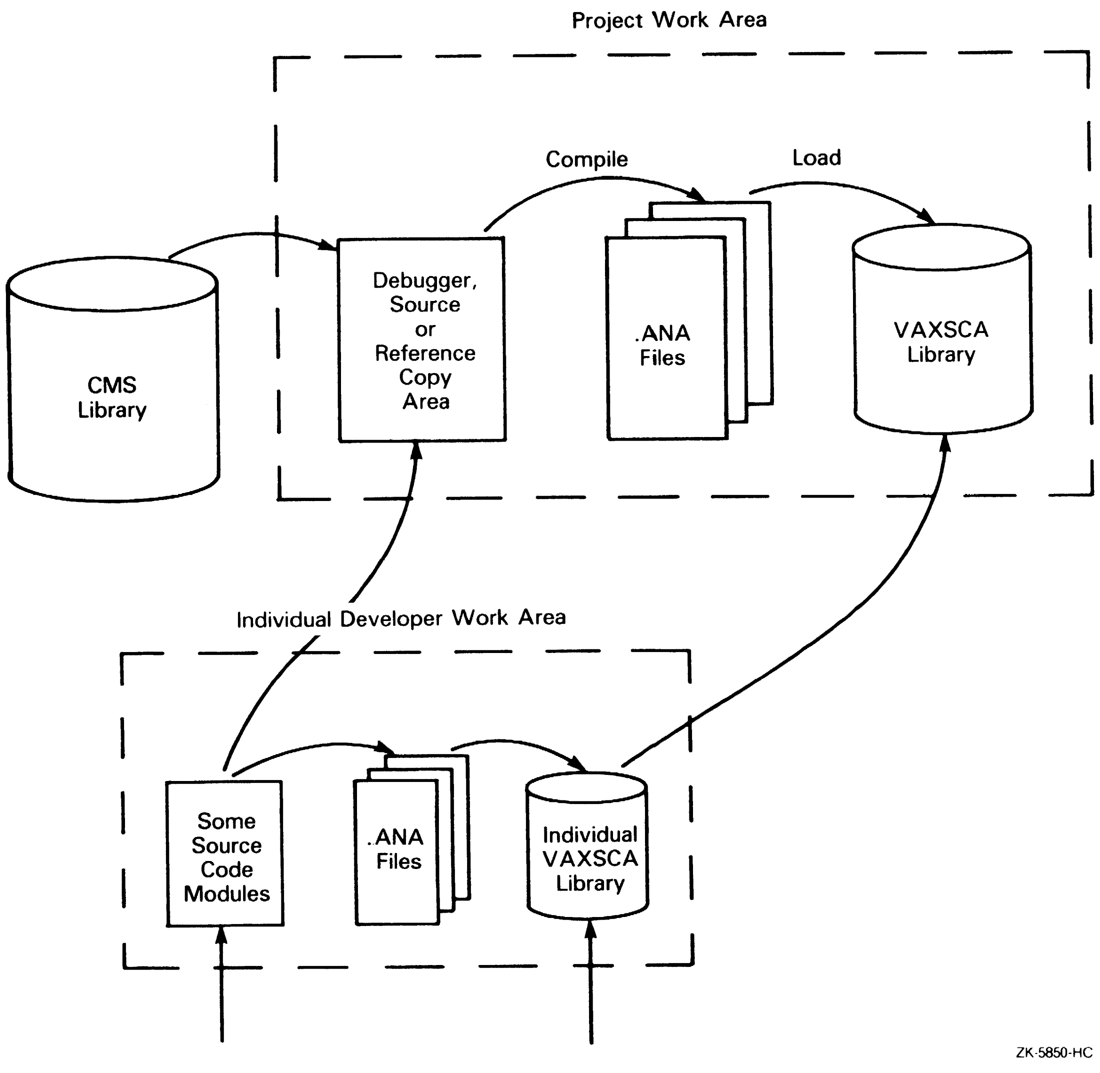
When a team member wants to do additional development work on specific modules, that member sets up an individual work area. Such individual work areas might consist of the following:

- Copies of source and object modules from the project libraries
- Local VAXSCA libraries that contain copies of the module information required to complete assigned tasks

To make available the module-viewing capabilities of VAXSCA/VAXLSE integration, the project team member must inform VAXLSE of the locations of latest sources, and the related source information. The team member provides pointers to these locations by supplying a search list for VAXLSE. The search list first points to source modules in individual team members' default directories, and then points to the remaining modules in the project source directory. With such an arrangement, each member can effectively "see" through the local work area to the project-wide area. If an individual work area contains only new modules, and all of the work can be done with local resources, the team member need not specify the pointers to the project-wide area.

The following sections provide a general overview of VAXSCA and discuss some of the commands that are available to you while using VAXSCA within VAXLSE. For detailed information on VAXSCA and its use with various programming languages, refer to the *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer.*

Figure F–1: Using VAXSCA for Multimodular Development



F.2.1 Setting up a VAXSCA Environment

To set up a VAXSCA environment, you must take the following steps:

- Create a VAXSCA library in a subdirectory.
- Select the library.
- Use the VAX PASCAL compiler to generate the data analysis (.ANA) files for each source module in your system.
- Load these data analysis files into your local VAXSCA library.

You are then ready to use VAXSCA to conduct source information queries.

F.2.1.1 Creating a VAXSCA Library

To use VAXSCA, you must have a VAXSCA library to store the detailed source analysis data that the VAX PASCAL compiler collects.

To create a VAXSCA library, establish a subdirectory at the DCL level. For example:

```
$ CREATE/DIRECTORY PROJ:[USER.LIB1]
```

This command creates a subdirectory LIB1 for a local VAXSCA library.

To initialize a new VAXSCA library, specify the CREATE LIBRARY command. This command has the following form:

```
   CREATE LIBRARY [/qualifier...] directory-spec[,...]
```

For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

This command initializes and activates library LIB1.

F.2.1.2 Generating Data Analysis Files

VAXSCA uses detailed source data that is generated by the VAX PASCAL compiler. When you specify the /ANALYSIS_DATA qualifier on the PASCAL command, the generated data is output to a file with the default type .ANA. For example:

```
$ PASCAL/LIST/DIAGNOSTICS/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.PAS, PG2.PAS and PG3.PAS, and generates four corresponding output files for each input file, with the file types OBJ, LIS, DIA and ANA. VAXSCA puts these files in your current default directory unless you specify otherwise.

F.2.1.3 Selecting a VAXSCA Library

To select an existing VAXSCA library to use with your current VAXSCA session, use the SET LIBRARY command. The command has the following form:

```
   SET LIBRARY [/qualifier...] directory-spec[,...]
```

A message appears in the message buffer, at the bottom of your screen, indicating whether your VAXSCA library selection succeeded.

F.2.1.4 Loading Data Analysis Files into a Local Library

Before you can examine the information in the compiler-generated source analysis (.ANA) files, you must load the files into a VAXSCA library using the LOAD command. The LOAD command has the following form:

```
   LOAD [/qualifier...] file-spec[,...]
```

For example:

```
LSE> LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1.ANA, PG2.ANA, and PG3.ANA.

F.2.2 Using VAXSCA for Cross-Referencing

With a VAXSCA library in place, you can ask for symbol or file information by using the VAXSCA command FIND. The FIND command has the following form:

```
  FIND [/qualifier...] [name-expression[,...]]
```

name-expression
Is the name of a symbol or file. It can be explicit or can contain wildcards.

For example:

```
LSE> FIND ABC,XY%
```

You can query VAXSCA library information for the following things that exist within a source program:

Name	A series of characters that uniquely identifies a symbol or a file
Item	An appearance of a symbol (such as a variable, constant, label, or procedure) or a file
Occurrence	The use of a symbol or a file

To limit the information resulting from a query, you can use qualifiers on the FIND command, such as the /DECLARATIONS and /REFERENCE qualifiers. For example:

```
LSE> FIND/REFERENCES=CALL BUILD_TABLE
```

This command causes VAXSCA to report only references in the source code where the routine BUILD_TABLE is called.

When you first issue a FIND command within VAXLSE, you initiate a *query session.* Within this context, the integration of VAXLSE and VAXSCA provides the following commands that can be used only within VAXLSE:

$\left\{\begin{array}{l}\text{NEXT}\\\text{PREVIOUS}\end{array}\right\}\left\{\begin{array}{l}\text{NAME}\\\text{ITEM}\\\text{OCCURRENCE}\\\text{QUERY}\\\text{STEP}\end{array}\right\}$	Closely-associated commands that let you step through one or more query buffer displays within VAXLSE.
GOTO SOURCE	Displays the source corresponding to the current query item.
GOTO DECLARATION	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window.

INDEX

VAX PASCAL
User Manual
AI–H485E–TE

READER'S COMMENTS

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

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

____
____
____
____
____
____

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

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________________

Street ______________________________________

City ______________________ State ____________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line

VAX PASCAL
User Manual
AI–H485E–TE

READER'S COMMENTS

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

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

____
____
____
____
____
____

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

____
____
____
____
____
____

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______________________

Name ______________________ Date ______________

Organization ______________________________________

Street ______________________________________

City ____________________ State __________ Zip Code or Country ________

Do Not Tear - Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698

Do Not Tear - Fold Here

Cut Along Dotted Line