digital

VAX–11 PL/I

User's Guide

Order No. AA-H951A-TE

VAX11

August 1980

Describes the operation of the VAX-11 PL/I compiler and the extensions to the PL/I language that support the execution of PL/I programs in the VAX/VMS operating system environment.

VAX-11 PL/I
User's Guide

Order No. AA-H951A-TE

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	VAX/VMS V2.0
SOFTWARE VERSION:	VAX-11 PL/I V1.0

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

digital equipment corporation · maynard, massachusetts

Contents

Chapter 6 Options of the ENVIRONMENT Attribute

Chapter 15 Global Symbols

Chapter 16 Return Status Values

Chapter 17 Error Signaling and Condition Handling

Part IV Programming Considerations and Examples

Chapter 18 Storage Allocation and Usage

Appendix B PL/I Messages

Appendix C Correspondence of PL/I and RMS

Appendix D ASCII Character Set

Index

Figures

Sample Programs

Preface

Manual Objectives

This manual describes how to use the VAX-11 PL/I compiler on the VAX/VMS operating system and contains detailed explanations of the extensions made to the standard PL/I language for VAX-11 PL/I. It includes information on the VAX/VMS commands and utilities to aid in program development, as well as information to assist in writing PL/I programs that take advantage of features of the file system and the operating system.

Audience Assumptions

This manual is designed for programmers who have a working knowledge of PL/I and some familiarity with the VAX/VMS operating system and its command language, DCL.

Structure of This Document

This manual has four parts:

- Part I, "The Command Language," provides information on the commands and utilities available for program development. This part provides information on compiling, linking, and executing VAX-11 PL/I programs on VAX/VMS.
- Part II, "The File System," provides specific information on the VAX-11 file system, RMS (Record Management Services). This part describes the relationship between PL/I input/output statements and VAX/VMS files, and provides detailed information on the ENVIRONMENT options of VAX-11 PL/I, I/O statement options, and file-handling built-in subroutines. This part provides examples of I/O to sequential, relative, and indexed sequential files and of file sharing.
- Part III, "Procedure Calling and Condition Handling," provides detailed information on calling procedures written in other languages from PL/I programs, including using system global symbols and return status values, and on condition handling.

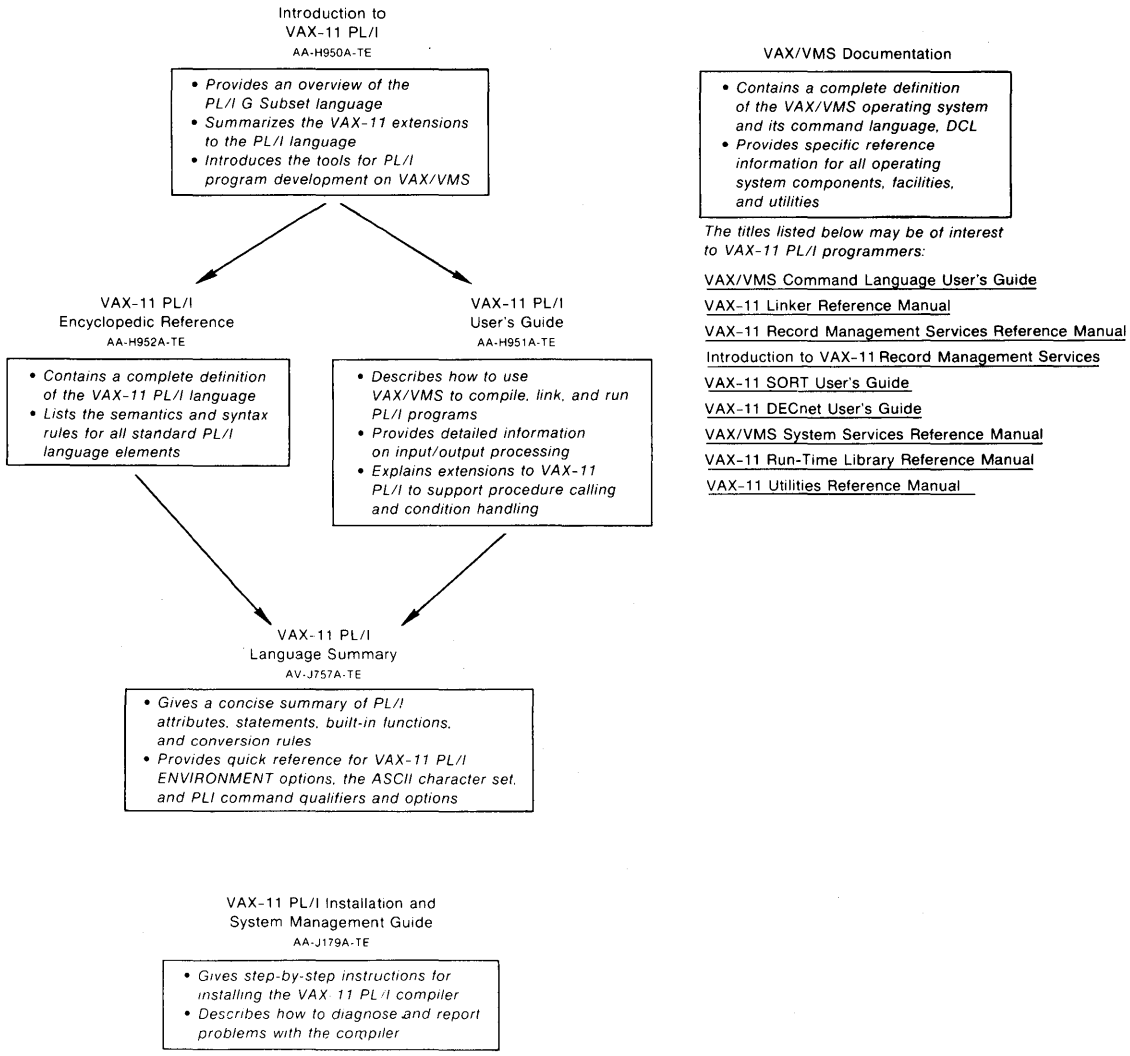


Figure 1: Documentation for VAX-11 PL/I Programmers

Part I
The Command Language

Chapter 1
Introduction to Program Development on VAX/VMS

The VAX-11 operating system, VAX/VMS, and its command language, DCL, provide numerous tools and utilities for program development. This chapter summarizes the basic things you need to know to use the command language in developing and testing your PL/I programs, including:

- The commands you use to create, compile, link, and execute PL/I programs
- The rules for specifying input and output files for commands and programs
- The commands available to you for file creation, modification, and maintenance

For a tutorial introduction to these concepts, see the *VAX/VMS Primer*. For detailed definitions of commands and file specifications, see the *VAX/VMS Command Language User's Guide*.

1.1 VAX/VMS Commands for Program Development

Figure 1-1 illustrates the DCL commands you use to create and run PL/I programs.

The commands are shown in their simplest forms. You can, however, specify qualifiers on these commands to request special processing or to indicate a special type of input file, as in these examples:

```
$ PLI/LIST=LP:  METRIC
$ LINK METRIC,MYLIB/LIBRARY
```

In this PLI command, the /LIST qualifier requests the compiler to create a listing file for the source program METRIC.PLI and to output it on a line printer device (LP: is the device name for line printers).

The LINK command uses the /LIBRARY qualifier to indicate that the input file MYLIB is a program library consisting of object modules. The linker will automatically search this library to locate external procedures and external variables that are referenced in the source file METRIC.PLI.

1.1.1 Hints for Entering Commands

The next few chapters of this manual describe in detail the commands of specific interest to PL/I programmers. You should note the following hints on entering commands:

- You can truncate (shorten) any command name or qualifier name to four characters. In some cases, fewer than four characters are accepted, as long as there is no ambiguity about the name of the command.
- You must precede each qualifier name with a single slash character (/).
- If you omit a required parameter, for example, a file specification, the DCL command interpreter will prompt you to enter it.
- You can enter a command on as many lines as you wish, as long as you end each continued line with a hyphen (-) character.
- After you have entered a complete command, you must press RET to pass the command to the system for processing.
- You can cancel a command before the final RET by using CTRL/Y.
- You can interrupt command execution by using CTRL/Y. To resume the interrupted command, enter the CONTINUE command. To stop processing completely after pressing CTRL/Y, you can begin entering other DCL commands.

If you make an error entering a command, for example if you misspell a command or qualifier name, the command interpreter issues an error message and you must reenter the entire command string.

1.1.2 HELP

You can obtain online information about a command, its parameters, or qualifiers by entering the HELP command. This command responds to a request for help on a command name by displaying a brief description of the command and by listing the additional information you can obtain. For example, if you enter the following:

```
$ HELP PRINT
```

The HELP command response displays a description of the PRINT command and a list of its qualifiers. To get further information, you must reenter the HELP PRINT command with an additional parameter: the name of the qualifier you want information about. For example:

```
$ HELP PRINT /JOB_COUNT
```

The HELP command responds to this request with a description of the syntax for entering the /JOB_COUNT qualifier.

The HELP command also provides detailed information about the PLI command and the VAX-11 PL/I language. You can obtain information about PL/I topics by specifying a PL/I keyword. For example:

```
$ HELP PLI INDEX
```

The HELP command responds to this request by displaying the syntax of the INDEX built-in function.

Table 1–1: Summary of File Specification Syntax

Field	Syntax Rules	Defaults	Notes
node	1 - 6 characters terminated by : :	local node	node: :node: : defines a path node"access-control": : *in VAX/VMS, username password* node: :"non-VMS-file-specification"
device dev c u	valid mnemonic or logical name A - Z 0 - 65535	SYS$DISK A 0	CR -card reader NET -network device DB -disk device MB -mailbox DM -RK06/7 disk MT -magnetic tape DX -floppy disk TT -terminal LP -line printer TU -cartridge tape
directory [name] [name.name...]	1 - 9 characters up to 8 names, separated by periods (.)	current default	[*] all directories [name...] all directories in path [*...] all subdirectories in all directories [-.name] back up a directory
filename	0 - 9 characters	*Input:* temporary defaults apply *Output:* same as input file	* — all file names *string* — match all names containing "string" str%ng — match any character in % position
filetype	0 - 3 characters preceded by .	Applied by command; temporary defaults apply	Wild card rules same as for filename **Command / Input / Output** PLI / PLI, TLB / OBJ, LIS LINK / OBJ, OLB / EXE, MAP LIBRARY / OBJ / OLB LIBRARY/TEXT / TXT / TLB RUN / EXE PRINT, TYPE / LIS
version	0 - 32767 preceded by ; or .	*Input:* highest *Output:* highest + 1	* — all versions ; — use most recent version

1.2.1 Temporary Defaults

Many VAX/VMS file-handling commands use temporary defaults under certain conditions. When a command such as PRINT or TYPE accepts a list of input file specifications, it uses explicit elements of one file specification as a temporary default for subsequent file specifications. Some examples follow:

```
$ PRINT [PROJECT.DATA]ALPHA,BETA.DAT,GAMMA
```

The PRINT command uses the default input file type LIS for the first input file and the file type DAT as specified for the second input file. It then applies the temporary default DAT to the file GAMMA. The PRINT command prints the highest existing versions of ALPHA.LIS, BETA.DAT, and GAMMA.DAT from the directory [PROJECT.DATA] on the current default device.

```
$ PRINT [PROJECT.DATA]FOREST.TXT,,DAT,,REF
```

Here, the PRINT command uses the temporary default FOREST as a file name and prints the files FOREST.TXT, FOREST.DAT, and FOREST.REF.

You must specify /GROUP on a DEFINE command to place a name in the group logical name table, and you must have the GRPNAM user privilege.

- System logical name table. There is a single system logical name table. The logical names in this table can be accessed by all users. You must specify /SYSTEM on a DEFINE command to place a name in the system logical name table, and you must have the SYSNAM user privilege.

1.2.3.1 Logical Name Translation — When the system attempts to locate an equivalence name for the name of a PL/I file, or for a portion of a file specification, it is said to be performing logical name translation. When the system translates a logical name, it searches the process, then group, then system logical name tables, in that order, for a logical name. Each time the system translates a logical name, it examines the result to see if it still contains a logical name. If it does, it translates the result. This recursive translation occurs until the file specification is complete or until ten recursive translations have been made.

You can determine the current equivalence for a logical name by entering the SHOW TRANSLATION command. For example:

```
$ SHOW TRANSLATION SRC
  SRC = [PROJECT.SRC]  (process)
```

The response gives the translation and indicates that the logical name SRC was found in the process logical name table.

A logical name assignment is deleted either when a new definition is given for the name or when the name is explicitly deleted with a DEASSIGN command. For example:

```
$ DEASSIGN SRC
```

This command deletes the logical name table entry for the logical name SRC.

1.2.3.2 Uses for Logical Names — VAX/VMS system programs use logical names in many ways. For example, the PL/I compiler and the linker use logical names to provide default libraries for INCLUDE modules and object module libraries, respectively. These uses of specific logical names are described in Sections 2.4.6, "Default PL/I Libraries," and 3.3.3, "Default User Object Module Libraries."

A principal use for programmers is to provide device and file independence for executable program images or command procedures. For example, the name you give a file constant in a PL/I source program can be a logical name: each time you execute the program, you can issue a DEFINE command to provide a different equivalence name for the PL/I file. The relationship between PL/I file constants and VAX/VMS file specifications is described in detail in Chapter 5, "Overview of the File System."

The use of logical names is not limited to file-related functions, however. For an example of using logical names to pass string arguments to PL/I main procedures, see Section 4.4.3, "Passing Data Via Logical Names."

Table 1-3: VAX/VMS Commands for File Maintenance

Category	Command	Command Function
File creation	CREATE	Creates a file from records or data that follows in the input stream; for example, lines entered from a terminal or placed in a batch input file.
	EDIT	Invokes one of the VAX/VMS interactive editing programs, for example, SOS, EDT, or EDI.
Correcting and modifying files	EDIT	Invokes one of the interactive editors to make changes or additions to a disk file.
Cataloging and organizing files	CREATE/DIRECTORY	Establishes a new directory or a hierarchy of directories to catalog files.
	DIRECTORY	Lists files and information about them. Can list files with common file names or file types, files in one or more directories, files created since a certain date, and so on.
	LIBRARY	Creates and maintains libraries of INCLUDE text modules and libraries of object modules.
	RENAME	Changes the directory in which a file is cataloged; or changes the file name, file type, or version number of a file or file.
	SET DEFAULT	Changes the current default device or directory.
Copying and backing up files	{ ALLOCATE / INITIALIZE / MOUNT }	Provide device-handling and control commands that let you access data written on nonsystem disks, on magnetic tapes, or on punched cards; or to output data to a disk or tape.
	COPY	Copies the contents of a file or files to another file or files.
Deleting files	DELETE	Makes the contents of a file inaccessible by removing its directory entry.
	PURGE	Deletes a specified number of earlier versions of a file or a group of files.

Chapter 2
Compiling PL/I Programs

This chapter describes how to use the PLI command to compile your source programs into object modules. It discusses:

- The functions of the compiler
- PLI command syntax and qualifiers
- Compiler diagnostic messages and error conditions
- INCLUDE files and libraries

2.1 Functions of the Compiler

The primary functions of the compiler are to verify the PL/I source statements and to issue messages if there are any errors; to generate machine language instructions from the source statements of the PL/I program; to group these instructions into units called program sections, and to write the program sections into an object module.

When it creates an object module, PL/I provides the linker with the following information:

- The module name. It takes this name from the name of the main procedure in the source program, that is, the procedure that specifies OPTIONS (MAIN). Note that this is not necessarily the name of the file containing the object module. If no procedure specifies OPTIONS(MAIN), the name of the object module is the name on the first procedure statement in the source file.
- A list of all entry points, external variables, and global symbols that are declared in the module. The linker uses this information when it binds two or more modules together and must resolve references to the same names in the modules.
- A summary of the program sections it has created and their attributes, the generated machine instruction text, and relocation information.

You must separate multiple input file specifications with either commas (,) or plus signs (+). The commas and plus signs have different meanings, as follows:

- Commas delimit PL/I source files to be compiled separately. PL/I compiles each file and creates an object module for each.
- Plus signs delimit files to be concatenated or libraries containing INCLUDE files. PL/I compiles the source files as a single file and creates one object module. Library file specifications must be qualified with the /LIBRARY qualifier.

If a file specification does not contain a file type, PL/I assumes a default file type of PLI for a source file. If a file specification is qualified with /LIBRARY, PL/I assumes a default file type of TLB. INCLUDE files and INCLUDE file libraries are described in Section 2.4, "INCLUDE Files."

A single file may contain more than one PL/I procedure; PL/I concatenates these procedures into a single object module as decribed in Section 2.2.3, "Concatenated Input Files."

Command Qualifiers

Command qualifiers request processing options of the compiler. You can specify qualifiers to the PLI command following the command name or following an individual file specification. When a qualifier is specified following the PLI command name, its action applies to each file in the list, unless overridden by a qualifier specified for an individual file.

When a qualifier is specified following a file specification in a list of files separated by commas, its action is applied only to the compilation of that file.

/CHECK
/NOCHECK

Controls the checking of array subscripts and of positional references in arguments to the SUBSTR built-in function. If you specify /CHECK, the compiler provides the following checks:

- It checks that each reference to the SUBSTR built-in function or pseudovariable lies within the string's current length.
- It checks that each reference to an array specifies subscripts that are within the bounds declared for the array.
- It checks that all string lengths are nonnegative and that all array extents are positive.

The default is /NOCHECK. /CHECK is primarily of use during initial program debugging; it results in the generation of additional code and, consequently, a slower program.

Table 2-1: PL/I Compiler Options

Option	Function
[NO]LIST_INCLUDE	Print/do not print the contents of INCLUDE files and modules in the program listing.
[NO]LIST_MAP	Print/do not print the storage map of the compiled program in the program listing. The storage map includes a list of all external entry points, the size and attributes of all variables that are referenced in the program, and a program section summary and procedure definition map.
[NO]LIST_SOURCE	Print/do not print the source program statements in the program listing.
[NO]LIST_STATISTICS	Print/do not print performance statistics in the program listing.

/G_FLOAT
/NOG_FLOAT

For VAX-11 computers that are equipped with the appropriate hardware option, specifies the default representation of floating-point variables with a precision in the range of 25 through 53.

By default, the compiler uses D (double-precision) floating point. Specify /G_FLOAT to override this default and to request the compiler to use the G floating-point type for these variables.

The default and maximum precisions for all floating-point formats are summarized in the *VAX-11 PL/I Encyclopedic Reference*.

/LIST[=**file-spec**]
/NOLIST

Controls whether a listing file is produced.

If the PLI command is executed from interactive mode, /NOLIST is the default, unless the /CROSS_REFERENCE or /MACHINE_CODE qualifiers are specified. If the PLI command is executed from batch mode, /LIST is the default.

When /LIST is in effect, the compiler gives a listing file the same file name as the source file and a file type of LIS.

If you specify a file specification with /LIST, the compiler uses that file specification to override the default values applied.

You can control the contents of the listing file by specifying the /CROSS_REFERENCE and /MACHINE_CODE qualifiers, and by specifying options on the /ENABLE qualifier.

/MACHINE_CODE
/NOMACHINE_CODE

Controls whether the listing file produced by the compiler includes a listing of the machine language code generated during the compilation.

For an example of a machine code listing, see Appendix A.

/WARNINGS
/NOWARNINGS

Controls whether the compiler prints messages for diagnostic warnings.

By default, the compiler prints all diagnostic messages during compilation. If you specify /NOWARNINGS to override this default, the compiler does not print warning messages. It does, however, continue to display messages for informational, error, and fatal diagnostics.

File Qualifier

/LIBRARY

Indicates that the associated input file is a library containing text modules that may be included in the compilation of one or more of the specified input files.

The specification of a library file must be preceded by a plus sign.

If the file specification does not contain a file type, PL/I assumes the default file type of TLB.

For more information on creating and using INCLUDE file libraries, see Section 2.4, "INCLUDE Files."

2.2.1 PLI Command Examples

```
$ PLI METRIC
```

The PLI command compiles METRIC.PLI and creates the file METRIC.OBJ.

```
$ PLI/ENABLE=LIST_INCLUDE/MACHINE_CODE APPLIC
$ PRINT APPLIC
```

The PLI command compiles the file APPLIC.PLI and creates the files APPLIC.OBJ and APPLIC.LIS. The listing shows the contents of all files and text modules included in the compilation by %INCLUDE statements, as well as a machine code listing of the program. The /LIST qualifier is not necessary because /MACHINE_CODE implies /LIST. The PRINT command queues a copy of the listing file for printing. Note that the default file type created by the compiler is LIS and that this is also the default file type assumed by the PRINT command.

```
$ PLI SWITCH,TXT/CHECK
```

The PLI command compiles the statements in the file SWITCH.TXT. The /CHECK qualifier causes the compiler to verify all array references and substring extents.

2.2.2 Specifying Input and Output Files

When you specify more than one input file on the PLI command, you can separate the names of the files with either commas or plus signs. If you separate them with commas, PL/I compiles each source file separately and creates individual listing files and object files for each.

In the third and fourth examples, the listing files are not saved on disk; they are deleted after output.

2.2.3 Concatenated Input Files

If you separate the names of input files with plus signs, PL/I concatenates the contents of the files and compiles them as if they were a single input file. It creates a single object file and (if /LIST is specified) one listing file for concatentated input files.

The rules in effect for compiling concatenated input files are the same as for a single file that contains more than one procedure. These are as follows:

- Only one procedure among all files that are to be concatenated may specify OPTIONS (MAIN); this procedure is the main entry point.
- PL/I gives the object module the same name as the first procedure in the file. It gives the object module output file the same file name as the first input file in the command line.
- If files contain separate level-one procedures, the procedures may call one another without declaration, but they may not reference internal variables declared within other blocks. (A level-one procedure is a procedure whose text is not contained within another procedure.)

For example, assume that the files A.PLI, B.PLI, and C.PLI have the following contents:

A.PLI contains:

```
A: PROCEDURE;
   DECLARE X FIXED BINARY;
   CALL B;
END;
```

B.PLI contains:

```
B: PROCEDURE OPTIONS (MAIN);
   .
   .
END;
```

C.PLI contains:

```
C: PROCEDURE;
   .
   .
END;
```

These files may be concatenated in a compilation as follows:

```
$ PLI A+B+C
```

This command causes PL/I to create the file A.OBJ that contains an object module named B; B is the main entry point. Within this module, procedures A, B, and C may invoke one another without declaration, but none of the procedures may refer to internal variables declared within the other. For example, B cannot reference the variable X declared within A.

Source file line number *n*

Specifies the source file line number of the statement that caused the error. Note that this line number is assigned to a statement by the compiler. It is not necessarily the same as the line number, if any, assigned by a text editing program.

The messages produced by the VAX-11 PL/I compiler are listed in Appendix B.

2.3.1 Suppressing Warning Messages and Parts of Messages

When you compile a PL/I program, you can use the /NOWARNINGS qualifier to request the compiler not to display warning (severity W) messages on the terminal. For example:

```
$ PLI METRIC/NOWARNINGS
```

When PL/I compiles the file METRIC.PLI, it does not display warning messages on the output device. You may find this qualifier useful when you are compiling programs that you know contain statements that cause warnings.

The DCL command SET MESSAGE lets you define whether you want to see messages displayed in their entirety or in shortened form. For example, if you do not want to see the %PLIG-s-ident part of messages, you can enter the command:

```
$ SET MESSAGE /NOFACILITY/NOSEVERITY/NOIDENT
```

This command cancels the facility, severity, and identification portion of all messages and remains in effect for all commands you subsequently enter, until you reissue the SET MESSAGE command or log off the system.

2.3.2 Interrupting the Compiler

During the compilation of a file, PL/I sometimes detects an error from which it cannot recover, that is, an error that causes additional errors to be detected. For example, a syntax error in a DECLARE statement causes subsequent references to the variables that were declared in that statement to generate errors.

When errors of this sort occur, you can halt the compilation, correct the errors, and restart the compiler. To do this, you can use the CTRL/C or CTRL/Y key combinations, according to the following guidelines:

- If you specified /LIST, and would like to examine the listing, you can press CTRL/C. The compiler will close the listing file it is creating. Although the file will not be complete (and may sometimes be empty), it often contains enough of the program listing, with diagnostic messages, that you can print it to determine which errors occurred and correct them.

2.4.1 Specifying INCLUDE Files

An INCLUDE file is requested by a %INCLUDE statement in a PL/I source file. When the compiler reads the %INCLUDE statement during compilation of a source program, it begins reading from the file specified by %INCLUDE. When it reaches the end of the included file, it resumes reading from the previous input file.

An INCLUDE file can contain a %INCLUDE statement. The maximum depth to which INCLUDE files can be nested is four.

The syntax for specifying %INCLUDE is:

$$\%\text{INCLUDE} \left\{ \begin{array}{l} \text{'file-spec'} \\ \text{text-module-name} \end{array} \right\} ;$$

file-spec
Is a file specification enclosed in apostrophes. The file specification can be any valid VAX/VMS file specification, including a logical name.

When the file specification does not completely specify the name of the INCLUDE file, PL/I uses the VAX/VMS system defaults for file specifications and uses the default file type of PLI.

text-module-name
Specifies the 1- to 31-character name of a text module in a library of INCLUDE files or other text modules. A module name can consist of the alphanumeric characters or the $ or _ characters. The name of the library containing the module must be specified on the PLI command.

For example, the following specifications are different:

```
%INCLUDE STATE;
%INCLUDE 'STATE';
```

In the first example, PL/I searches any library files specified on the PLI command for a module named STATE. In the second example, PL/I assumes that STATE is a file specification and looks for the file STATE.PLI in the current default directory.

If PL/I cannot locate a specified file or module, it issues a fatal error message and terminates the compilation.

2.4.2 Text Libraries

A text library is a file that contains individual files and a table indexing them. The LIBRARY command creates and modifies text libraries; these libraries have a default file type of TLB. To use libraries for PL/I INCLUDE files, you must:

1. Create one or more libraries consisting of INCLUDE files.

2. Specify the name of the INCLUDE module in a %INCLUDE statement in the PL/I source program.

3. Specify the name of the library on the PLI command to compile the source program or define a default library.

This command inserts the contents of the file DECLARE.PLI into the library PLIFILES under the name EXTERNAL_DECLARATIONS. This module can be included in a PL/I source file during compilation with the statement:

```
%INCLUDE EXTERNAL_DECLARATIONS;
```

Table 2–4 summarizes the commands that create libraries and provide maintenance functions. For a complete list of the LIBRARY command qualifiers, and for a description of other DCL commands listed in Table 2–4, see the *VAX/VMS Command Language User's Guide.*

Table 2–4: Commands to Control Library Files

Function	Command Syntax[1]
Create a library.	$ LIBRARY/TEXT/CREATE *library-name file-spec,...*
Add one or more modules to a library.	$ LIBRARY/TEXT/INSERT *library-name file-spec,...*
Replace one or more modules in a library	$ LIBRARY/TEXT/REPLACE[2] *library-name file-spec,...*
Specify the names of modules to be added to a library.	$ LIBRARY/TEXT/INSERT *library-name file-spec*/MODULE=*module-name*
Delete one or more modules from a library.	$ LIBRARY/TEXT/DELETE=*(module-name,...) library-name*
Copy a module from a library into another file.	$ LIBRARY/TEXT/EXTRACT=*module-name library-name*
List the modules in a library.	$ LIBRARY/TEXT/LIST/OUTPUT=*file-spec library-name*
Rename a library or move a library to another directory.	$ RENAME *old-library-name new-library-name*
Delete a library.	$ DELETE *library-name*
Copy or backup a library.	$ COPY *input-library-name output-library-name*

1. The LIBRARY command qualifier /TEXT indicates a text module library. By default, the LIBRARY command assumes an object module library.
2. REPLACE is the default function of the LIBRARY command, if no other action qualifiers are specified. If no module exists with the given name, /REPLACE is effectively /INSERT.

2.4.4 Specifying Library Files on the PLI Command

When you specify a library file on a PLI command, you must precede the file specification of the library with a plus sign and use the /LIBRARY qualifier. For example:

```
$ PLI APPLIC+DATAB/LIBRARY
```

You can define the logical name PLI$LIBRARY in the process, group, or system logical name table. If the name is defined in more than one table, the PL/I compiler uses the equivalence for the first match it finds in the normal order of search (that is, the process, then group, then system table). Thus, if PLI$LIBRARY is defined in both the process and group logical name tables, the process logical name table assignment overrides the group logical name table assignment.

2.4.7 Default System INCLUDE Library

When it cannot find INCLUDE modules in libraries specified on the PLI command or in the default library defined by PLI$LIBRARY, PL/I searches the library identified by the following name:

```
SYS$LIBRARY:PLISYSDEF.TLB
```

Where SYS$LIBRARY is normally defined by the system manager to identify the device and directory containing system libraries. PLISYSDEF.TLB is a library of INCLUDE modules supplied by VAX-11 PL/I. It contains declarations for the entry points for VAX/VMS system services, local symbol definitions required for use with system services, and variables to test return status values from system services. The contents of this library are described in detail in Chapter 19, "System Services."

Note that you can define a second default private library by redefining the logical name SYS$LIBRARY. For example, if you make a copy or subset of PLISYSDEF.TLB or create your own library named PLISYSDEF.TLB, you can make this library the second search library as follows:

```
$ DEFINE PLI$LIBRARY DB1:[MALCOLM.LIBRARY]STATEDATA.TLB
$ DEFINE/USER SYS$LIBRARY DB1:[MALCOLM.LIBRARY]
$ PLI MAILBOX
```

In this example, the DEFINE command creates process logical name table assignments for PLI$LIBRARY and SYS$LIBRARY. When PL/I compiles the program MAILBOX, it searches STATEDATA.TLB and then PLISYSDEF.TLB in DB1:[MALCOLM.LIBRARY] for any INCLUDE modules that are specified in MAILBOX. The /USER qualifier on the second DEFINE command ensures that the logical name SYS$LIBRARY will be deassigned following the execution of the PLI command. This is necessary because other system programs (the linker, for example) that use SYS$LIBRARY may not execute correctly if the library is not reassigned. For additional information on logical names, see Section 1.2.3, "Logical Names."

Chapter 3
Linking Programs

This chapter describes how to use the linker and object module libraries to combine object modules into executable programs. It discusses:

- The functions performed by the linker
- The LINK command and its input and output files
- Creating and using object module libraries

The topics in this chapter are confined to areas of particular interest to PL/I programmers. For additional information on linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VAX-11 Linker Reference Manual.*

3.1 Functions of the Linker

The primary functions of the linker are to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation.

For any PL/I procedure, the object module generated by the compiler contains calls and references to VAX-11 PL/I run-time procedures. For example, any procedure that contains a PUT statement requires calls to routines to convert noncharacter data to a character string and calls to I/O routines.

These run-time procedures are automatically located in the default system object module libraries. These libraries are described in more detail in Section 3.3.4, "System Libraries."

A global symbol can be any of the following:

- The name of an external procedure or entry point. In PL/I terms, this is the name specified on an ENTRY statement or on a PROCEDURE statement in a level-one procedure, that is, a procedure at the outermost level whose text is not contained within the text of any other procedure.
- The name of a variable declared with the EXTERNAL STATIC attributes.
- The name of a variable declared with the GLOBALDEF or GLOBALREF attribute. Variables of this type are described in Chapter 15, "Global Symbols."

Table 3–1: LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output files and define a file specification.	/EXECUTABLE[=*file-spec*] /SHAREABLE[=*file-spec*] /SYMBOL_TABLE[=*file-spec*]	/EXECUTABLE=*name*.EXE, where *name* is the name of the first input file. /NOSHAREABLE /NOSYMBOL_TABLE
Request and specify the contents of a memory allocation listing.	/BRIEF /[NO]CROSS_REFERENCE /FULL /[NO]MAP	/NOCROSS_REFERENCE /NOMAP (interactive) /MAP=*name*.MAP (batch) where *name* is the name of the first input file.
Specify the amount of debugging information.	/[NO]DEBUG /[NO]TRACEBACK	/NODEBUG /TRACEBACK
Indicate that input files are libraries and to specifically include certain modules.	/INCLUDE=(*module-name*....) /LIBRARY /SELECTIVE_SEARCH	
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB /[NO]SYSSHR /[NO]USERLIBRARY[=*table*]	/SYSLIB /SYSSHR /USERLIBRARY=ALL
Indicate that an input file is a linker options file.	/OPTIONS	

3.2.1 Linker 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 normally need additional information to determine the specific error. Some of the more 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 had warnings or errors during compilation. You can often link compiled modules for which the compiler flagged warnings, but you should verify that the modules will actually produce the output you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one module has an entry point designated with the OPTIONS (MAIN) keywords. The image file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.

This LINK command links the object module METRIC.OBJ with the modules PRINTLINE and TERMLINE from the library FORMATS.OLB. Any references to external procedures and variables that are not defined in any of these three modules will cause the linker to search the library MATHLIB.OLB, in the directory [PROJECT.OBJLIB], before it searches the system libraries.

The format and content of a linker options file are described in detail in the *VAX-11 Linker Reference Manual*. You may wish to use an options file if you have a very long list of input files to specify, if you want to link a module with a shareable image file, or if you want to request special linker options.

When you specify more than one input file for the LINK command, the linker combines individual object files or modules explicitly included from a library in the order in which they are listed.

3.2.3 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 image file has the same file name as the first, or only, object module specified and a file type of EXE. For example:

```
$ LINK A,B,C
```

This LINK command links the object modules in the files A.OBJ, B.OBJ, and C.OBJ and creates the image file A.EXE.

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

Some of the rules for naming input and output files are shown in Table 3-2. These rules apply to the specification of names with the /MAP qualifier as well.

Table 3-2: Specifying Input and Output Files for the Linker

Rule	Example	Output File(s)
If you do not specify the /EXECUTABLE qualifier, the linker gives the image file the same name as the first input file.	$ LINK METRIC	METRIC.EXE
If you specify /EXECUTABLE following the name of an input file, the linker uses that file's name to name the output file.	$ LINK METRIC,APPLIC/EXECUTABLE	APPLIC.EXE
If you give a file specification with the /EXECUTABLE qualifier, the linker uses that file specification.	$ LINK/EXECUTABLE=TEST METRIC,APPLIC	TEST.EXE
When you specify a device and/or directory for a file specification, that device and/or directory becomes a temporary default for the remaining input and output files.	$ LINK METRIC,- $_[PROJECT.OBJLIB]MATHLIB/LIBRARY,- $_FORMATS /EXECUTABLE	[PROJECT.OBJLIB]FORMATS.EXE

3.2.5 Specifying Debugging Qualifiers

You can specify either the /DEBUG or /TRACEBACK qualifiers when you link an image. These qualifiers control the amount of debugging information that is available to the VAX-11 Symbolic Debugger and to the run-time error reporting mechanism.

By default, the linker includes traceback information. This information lets the run-time system list all of the active procedure invocations at the time of a fatal run-time error. If you specify the /NOTRACEBACK qualifier, that information will not be available.

Regardless of whether you specified /DEBUG to the PL/I compiler, you can specify /DEBUG when you link the object module. This qualifier requests that the object modules containing the debugger program be linked to your object modules. When you execute the program, the debugger initially takes control. The steps required to run a program under the control of the debugger and the symbolic debugging capabilities available for PL/I programmers are described in the *VAX-11 PL/I Guide to Program Debugging*.

3.3 Object Module Libraries

An object module library is a single file containing individual object modules and two tables that index the modules:

1. A module name table lists the names of the modules in the library. These are the names given to the modules when they are compiled.
2. A global symbol table lists all global symbols defined in each module.

These are the tables that are searched by the linker.

3.3.1 Creating an Object Module Library

The LIBRARY command creates and updates libraries. It assumes by default that a library upon which it is performing a function is an object module library. You can use object module libraries to:

- Catalog and group together commonly used subroutines and functions.
- Provide a default set of modules for the linker to use in resolving global references in object modules it is linking.
- Enhance the performance of linking operations by putting all needed modules in a single library, thus reducing the number of files that need be opened during the linking.

Figure 3-2 illustrates the sequence of creating object modules, creating a library, and using the library in linking programs.

The LIBRARY command uses the following default file types:

- An object module library file is assumed to have a file type of OLB.
- An object module file is assumed to have a file type of OBJ.

When the LIBRARY command inserts an object module in a library, it:

- Enters the name of the module in the library's module name table.
- Enters all global symbols from the object module, including the names of all entry points and all variables designated as global symbols, in the library's global symbol table.

For example, a PL/I program named QUEUES.PLI might contain the following designations:

```
READY: PROCEDURE;
      .
      .
ADDEL: ENTRY (QUEUE,POINTER);
      .
      .
REMVEL: ENTRY (QUEUE,POINTER);
```

This module can be compiled and placed in a library as follows:

```
$ PLI QUEUES
$ LIBRARY/INSERT DEFLIB QUEUES
```

After this LIBRARY command, the module name table for the library DEFLIB.OLB contains an entry for the module named READY. The library's global symbol table contains entries for the names ADDEL, REMVEL, and READY. Object modules that refer to any of these names can be linked with this library. When the library is specified as input to the linker, the linker searches the library's module name table and global symbol table for unresolved references.

3.3.2 Defining the Search Order for Libraries

When you specify libraries as input for the linker, you can specify as many as you wish; there is no practical limit. More than one library can contain a definition for the same module name. The linker uses the following conventions to search libraries specified in the command string:

- A library is searched only for definitions that are unresolved in the previous input files specified.
- If more than one library is specified for an object module, the libraries are searched in the order in which they are specified.

For example:

```
$ LINK METRIC,DEFLIB/LIBRARY,APPLIC
```

2. The process, then group, and then system logical name tables are searched for the name LNK$LIBRARY_1. If the logical name exists in any of these tables, and if it contains the desired reference, the search is ended.

This search sequence is taken for each reference that remains unresolved.

The search order can be modified for a particular link operation. To override the search of a library, you can do one of the following:

- Delete the logical name assignment for the library you do not want searched. For example:

```
$ DEASSIGN LNK$LIBRARY
```

  The DEASSIGN command deletes the logical name table entry LNK$LIBRARY.

- Specify /USERLIBRARY or /NOUSERLIBRARY on the LINK command. These qualifiers let you specify the PROCESS, GROUP, and SYSTEM keyword options to explicitly control which logical name tables are to be searched for default user libraries. For example:

```
$ LINK/USERLIBRARY=GROUP input-file,...
```

  When it executes this command, the linker searches only the GROUP logical name table. Specify /NOUSERLIBRARY to exclude all default user libraries in the search.

For complete details on defining and using default user libraries, see the *VAX-11 Linker Reference Manual.*

3.3.4 System Libraries

The directory identified by the system-defined logical name SYS$LIBRARY contains the library files:

- VMSRTL.EXE
- STARLET.OLB

The file VMSRTL.EXE contains the VAX-11 Run-Time Library. The procedures in this library provide:

- Commonly used mathematical and string-handling functions
- Procedures that support code produced by VAX/VMS compilers

VMSRTL.EXE is a library in shareable image format; that is, it is prelinked and can be accessed by many images concurrently. The procedures in a shareable image library can be used by a program even though the procedures are not physically included in the program image; the references to the procedures in the shareable image library are not resolved until the program is actually run. For information about creating shareable image libraries, and a description of the VAX-11 Run-Time Library, see the *VAX-11 Run-Time Library Reference Manual.*

STARLET.OLB contains, in object module form, all the procedures in VMSRTL.EXE, as well as additional run-time modules required by various

Chapter 4
Running PL/I Programs on VAX/VMS

This chapter describes the following considerations for executing your PL/I programs on the VAX/VMS operating system.

- Using the RUN command to execute programs interactively
- Passing status values to the command interpreter
- Executing programs in command procedures and batch jobs
- Passing arguments to a PL/I program from a DCL command string

For further information on any of the DCL commands or topics presented in this chapter, see the *VAX/VMS Command Language User's Guide*.

4.1 The RUN Command

You execute a PL/I program with the RUN command. The RUN command assumes by default that the file type of a program image is EXE, so you need not specify it. For example:

```
$ RUN METRIC
```

This RUN command locates the file METRIC.EXE in the current default directory. It then gives control to the main entry point, that is, the entry point designated with the OPTIONS (MAIN) keywords on its PROCEDURE or ENTRY statement. If no procedure specifies OPTIONS(MAIN), then control is given to the first, or only, module in the image.

4.1.1 Image Exit

When the main procedure executes a RETURN or END statement, or when any procedure in the program executes a STOP statement, the image is terminated. In the context of the VAX/VMS operating system, the termination of an image, or image exit, causes the system to perform a variety of clean-up operations during which open files are closed, system resources are freed, and so on.

Table 4–1: Effects of FINISH Condition

Procedure Has MAIN Option	Procedure Has FINISH ON-unit	Action When Image Exit Occurs[1]
yes	yes	The FINISH ON-unit is executed. If the ON-unit does not execute a nonlocal GOTO, the program terminates after the ON-unit executes.
yes	no	The program terminates normally.[2]
no	yes	The FINISH ON-unit is executed. If the ON-unit does not execute a nonlocal GOTO, the program terminates following the ON-unit.
no	no	The image exits.[2]

1. If the FINISH condition is signaled by the PL/I default condition handler following an ERROR condition, no FINISH ON-unit is executed.
2. If the program is executed under control of the VAX-11 Symbolic Debugger, the Debugger displays the PL/I FINISH message.

If there is a default PL/I condition handler, the message describes the error that occurred in PL/I terms. Otherwise, the message describes the error in VAX/VMS system terms.

In either case, the message is followed by a traceback. For each module that has traceback information, the default handler lists the procedures that were active when the error occurred and the sequence in which the procedures were called, that is, the order of block activation.

For example, if an integer divide-by-zero condition occurs, and no ON-unit for this condition exists in any active procedure block, the following run-time messages appear:

```
%PLI-F-ERROR, PL/I ERROR condition signaled
%SYSTEM-F-FLTDIV, arithmetic trap, floating/decimal divide
by zero at PC=000007C4, PSL=03C000A5
```

This message is followed by a traceback message as in the following example:

```
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name   line   relative PC  absolute PC

 SETUP           DIVIDE          9     00000074     000007C4
 SETUP           BEGIN%4         4     00000035     00000707
 SETUP           SETUP           4     0000000C     000006D0
 LIBS            NEXT           14     00000044     000006A3
 LIBS            LIBS           15     0000004C     0000065E
```

These columns provide information as described below.

module name

Indicates the name of a level-one procedure that was active when the error occurred.

The first module name is the name of the module in which the error occurred. Each subsequent line gives the name of the caller of the procedure

This command interrupts the program APPLIC. After you have interrupted a command, you can cause it to terminate by entering a DCL command that causes another image to be executed or by entering the DCL command EXIT. In this case, PL/I signals the FINISH condition to allow a FINISH ON-unit to execute before the given DCL command is executed.

Following a CTRL/Y interruption, you can also force an entry to the debugger by entering the DEBUG command. The debugger is described in the *VAX-11 PL/I Guide to Program Debugging*.

There are some other DCL commands you can enter that have no direct effect on the image. You can use these commands and resume the execution of the image with the DCL command CONTINUE. For example:

```
$ RUN APPLIC
^Y
$ SHOW TRANSLATION INFILE
  INFILE =     (undefined)
$ DEFINE INFILE DBA1:[TESTFILES]JANUARY.DAT
$ CONTINUE
```

In this example, CTRL/Y interrupts the image APPLIC. The SHOW TRANSLATION command indicates that no equivalence name is present for the logical name INFILE. The DEFINE command establishes an equivalence for the name and the CONTINUE command resumes the program.

For a complete list of the commands you can enter following a CTRL/Y interruption without affecting the current image, see the *VAX/VMS Command Language User's Guide*.

In most cases, the effect of CTRL/C and CTRL/Y on a program is the same. However, some programs (including programs you may write) establish particular actions to take to respond to CTRL/C. If a program has no CTRL/C handling routine, then CTRL/C is the same as CTRL/Y and in fact is echoed as ^Y on the terminal. For information on writing a CTRL/C routine for a PL/I program, see Section 19.4.4, "A CTRL/C Handling Routine."

4.2 Returning Status Values to the Command Interpreter

You can define a main procedure to be executed under the control of the DCL command interpreter as a PL/I function. Then the RETURN statement can specify a status value to be used as a success, failure, or informational indicator to the command interpreter. For example:

```
TESTP: PROCEDURE OPTIONS (MAIN) RETURNS (FIXED BINARY(31));
   .
   .
   RETURN (value);
```

where the value specified on the RETURN statement can be any constant, variable, or expression that can be converted to a fixed-point binary value. For meaningful results, you must specify the returns descriptor on the RETURNS option for the PROCEDURE statement as FIXED BINARY (31).

When any program or command is executed under the control of the DCL command interpreter, the general register R0, by convention, indicates the

4.3 Using Command Procedures

A command procedure is a file that contains a sequence of VAX/VMS commands and, optionally, data. You can cause the commands in the procedure to be executed in either of two ways:

- Interactively, you specify the name of the file following the @ (Execute Procedure) command. For example:

```
$ @TESTAM
```

  The @ command assumes that the file type of a command procedure is COM. This command executes the procedure TESTAM.COM.

- You can submit the command procedure to a system batch job queue for execution. After the job completes, the system prints a log file that indicates how the job ran. The SUBMIT command submits a job. For example:

```
$ SUBMIT TESTAM
```

  This command enters the file TESTAM.COM to the system batch job queue.

You can devise and use command procedures to simplify and enhance your program development. For example, you can write a command procedure that will compile, link, and run a specific PL/I program. The command procedure can specify all the needed libraries for the PLI and LINK commands, and can even contain all the input data you would require to test the program.

Command procedures can also be generalized. By taking advantage of DCL commands such as the assignment statement, and the IF, GOTO, and ON commands, you can write a command procedure that looks like a PL/I program: it can process variables, make decisions based on their values, and perform error condition handling.

The example in Figure 4–1 shows a simple procedure that will give you an idea of how to construct and use command procedures to help you with your PL/I program development and testing.

6. The program APPLIC is executed. It reads input data from the default input device. When a command procedure executes, the default input device is the command procedure itself. Thus, the data is read from the procedure file. In a command procedure, any line that does not begin with a dollar sign is treated as input data for the previous command or program. Program input terminates (and an actual end-of-file condition occurs) when a line that begins with a dollar sign is read. In this example, the program APPLIC reads all the lines between the RUN command and the EXIT command.

For more detailed information on the commands shown above, and for additional examples of techniques you can use in command procedures, see the *VAX/VMS Guide to Using Command Procedures*.

4.4 Passing Data to a Main Procedure

The VAX/VMS command interpreter does not provide an explicit interface for passing arguments to a main program. There are, however, programming techniques that permit you to pass data at run time to affect the execution of a program. This section describes the following techniques:

- Defining the program as a foreign DCL command and specifying data on the command line that invokes the program
- Using logical names as program arguments

Both of these techniques are restricted to character-string arguments.

The examples in the sections that follow provide enough information for you to duplicate them. Where appropriate, the text refers to additional required conceptual information found elsewhere in this manual.

4.4.1 Passing Data from the Command Line

When any command is executed on the VAX/VMS system, the command interpreter stores a copy of the command in an internal buffer. You can access the data in this buffer from a PL/I program by coding a call to a run-time library procedure named LIB$GET_FOREIGN. To use this routine, there are two things you must do:

1. To pass data to a program when it is run, you must use a DCL assignment command to equate a command symbol to a foreign command name. The command name will be used instead of a RUN command to execute the program image.
2. To obtain the data, the program must call a run-time library procedure to obtain the command string and must perform all string parsing and analysis itself.

Each of these mechanisms is described in detail on the following pages.

Sample Program 4-1: Obtaining the Command Line from the Command Interpreter

```
GETLINE: PROCEDURE OPTIONS(MAIN);

%INCLUDE $STSDEF;❶

DECLARE LIB$GET_FOREIGN EXTERNAL ENTRY (❷
                CHARACTER(*),   /* BUFFER TO RECEIVE COMMAND STRING */
                CHARACTER(*),   /* PROMPT STRING, IF ANY */
                FIXED BIN(15))  /* LENGTH OF COMMAND STRING */
        RETURNS (FIXED BINARY(31));

DECLARE COMMAND_STRING CHARACTER(500),  /* OUTPUT BUFFER */
      ❸ RETURN_LENGTH FIXED BINARY(15), /* OUTPUT LENGTH */
        PROMPT_STRING CHARACTER(12) STATIC  INITIAL('Enter line: ');

        STS$VALUE = LIB$GET_FOREIGN(COMMAND_STRING,
      ❹                             PROMPT_STRING,
                                    RETURN_LENGTH);
        IF ^STS$SUCCESS THEN GOTO ERROR;
      ❺
      ❻ PUT SKIP LIST(SUBSTR(COMMAND_STRING,1,RETURN_LENGTH));
        RETURN;

ERROR:
        PUT SKIP LIST('ERROR IN LIB$GET_FOREIGN');
        RETURN;
END;
```

The following notes are keyed to Sample Program 4-1:

1. The procedure includes the module $STSDEF from the default INCLUDE library PLISYSDEF.TLB. This module contains the declarations for the variables STS$VALUE and STS$SUCCESS, which are used to test whether the procedure LIB$GET_FOREIGN completed successfully.

2. LIB$GET_FOREIGN has three parameters: an output buffer, an optional prompt string, and a variable to receive the length of the string returned in the output buffer.

   These parameters are declared in the entry declaration. The parameter descriptors for the character-string parameters must be declared as CHARACTER(*).

3. The corresponding arguments for the procedure's three parameters are COMMAND_STRING, PROMPT_STRING, and RETURN_LENGTH, respectively.

4. LIB$GET_FOREIGN is invoked as a function so that its return value can be tested. The function reference returns the status into the variable STS$VALUE. The variable STS$SUCCESS is a one-bit field based on the low-order bit of STS$VALUE. This bit, if true, indicates a successful return.

Logical names and equivalence names are each limited to 63 alphanumeric characters, including dollar signs ($) and underline characters (_). Uppercase and lowercase letters are not equivalent. For example, the following commands would result in different equivalence names:

```
$ DEFINE NAME MABEL
$ DEFINE NAME "Mabel"
```

Note that the command interpreter translates all strings that are not enclosed in quotation marks to uppercase.

You can specify numeric character strings for integer arguments. For example, to define an equivalence name for the logical name NUMBER_OF_ITERATIONS, you could specify:

```
$ DEFINE NUMBER_OF_ITERATIONS 10
```

Note that the program that translates this name must perform an explicit conversion of the character-string value to an arithmetic data item.

4.4.2.2 Translating Logical Name Arguments — A procedure that interprets a logical name argument must explicitly translate the logical name by invoking the SYS$TRNLOG system procedure. This procedure is in the default system object module library, and will be automatically located when you link a program that references it.

This procedure has the following parameters:

1. A CHARACTER(*) parameter that represents the logical name string to be translated.
2. A FIXED BINARY(15) parameter in which the procedure places the length of the resultant equivalence name string.
3. A CHARACTER(*) parameter in which the procedure places the translated equivalence name string.
4. Three optional parameters for which the procedure provides default values. These parameters are of no interest in this example and may be omitted.

The program PRINT_NAME in Sample Program 4–2 illustrates a main procedure that translates the logical names NAME and NUMBER_OF_ITERATIONS and uses the resulting equivalence name strings.

The following notes are keyed to Sample Program 4–2:

1. The procedure includes the INCLUDE modules SYS$TRNLOG and $STSDEF. These modules are in the PL/I default INCLUDE library PLISYSDEF.TLB; they contain the declarations of the SYS$TRNLOG system service and the variables STS$VALUE and STS$SUCCESS, which are used to test whether the system service call completed successfully.

2. For each logical name, the procedure declares a string initialized to the logical name value, an output string variable, and an output length variable.

3. The integer variable ITERATION_COUNT is used in the conversion of the character-string equivalence name returned for the logical name NUMBER_OF_ITERATIONS.

4. SYS$TRNLOG is invoked as a function so that its return status can be tested.

5. The trailing commas in the argument list indicate arguments that are not specified — the SYS$TRNLOG system service provides default values for these arguments. The commas must, however, be specified. For details on omitting optional arguments to procedures, see Section 14.5.2, "Optional Arguments."

6. The function reference returns the status into the variable STS$VALUE. The variable STS$SUCCESS is a one-bit field based on the low-order bit of STS$VALUE. This bit, if set to 1, indicates a successful return.

   If either call to SYS$TRNLOG is not successful, the procedure exits. For complete details on testing return status values, and a description of the variables STS$VALUE and STS$SUCCESS, see Chapter 16, "Return Status Values."

7. The variable ITERATION_COUNT is assigned a value using the SUBSTR built-in function, which extracts the valid portion of the equivalence string from the 63-character equivalence name. This value is used to control the execution of the DO-group that follows.

   Note that the SYS$TRNLOG procedure returns a successful status if a logical name is not defined; thus, the results are unpredictable (and usually cause an error) if the procedure does not explicitly test the return status to see if a name was translated.

Part II
The File System

Chapter 5
Overview of the File System

This chapter introduces aspects of the VAX/VMS operating system that relate to PL/I input and output. It describes:

- The relationship between PL/I input/output statements and VAX/VMS input/output procedures
- File-naming and definition conventions, including a description of VAX/VMS logical names and process permanent files
- File system error handling at run time
- ENVIRONMENT options for input/output optimization

5.1 Relationship of PL/I I/O to the VAX/VMS File System

When a PL/I program contains an input/output statement, for example, OPEN or READ, the compiler translates the request into a call to the appropriate VAX/VMS operating system procedure.

In the VAX/VMS system, input/output is performed by:

- VAX-11 Record Management Services (RMS). RMS provides complete file and record-handling capabilities.
- Input/output system services. System services provide direct control over data transfer between the process executing an image and a peripheral device.

Note that although it is possible to call RMS procedures and VAX/VMS system services directly from a PL/I program, it is not normally necessary to do so. A PL/I program executed on the VAX/VMS operating system has full access to RMS capabilities through:

- Options of the ENVIRONMENT attribute
- Keyword options on PL/I input/output statements
- Built-in subroutines that invoke RMS file-handling services

RMS, in turn, manages the details of communicating with the VAX/VMS I/O system to effect data transfer and to organize and arrange data on physical devices.

2. It supplies missing fields from the value specified in the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute, if that option is specified.
3. It then applies system defaults to complete the file specification.

If the file specification that is finally achieved is invalid (for example, if it contains a dollar sign or underline character) or represents an illegal device or file (for example, if an input file cannot be found), the UNDEFINEDFILE condition is signaled.

The actions that PL/I and the file system take for each of these steps are described in more detail in the following sections.

5.2.2 Using Logical Names

At the command level before executing a program, you can create a logical name to assign a VAX/VMS file specification to the identifier of a PL/I file constant or to a value specified in a TITLE option. For example, suppose a PL/I program declares and opens a file as follows:

```
DECLARE INFILE FILE;
    .
    .
OPEN FILE (INFILE) RECORD INPUT;
```

You associate a VAX/VMS file with the identifier INFILE as in this example:

```
$ DEFINE  INFILE  DB1:[TEMP]A.DAT
```

The DEFINE command gives the PL/I file INFILE the VAX/VMS file equivalent of DB1:[TEMP]A.DAT. In VAX/VMS terms, the name INFILE is a logical name and the name DB1:[TEMP]A.DAT is an equivalence name for the logical name.

You can also use the DEFINE command to specify alternate device or file equivalents for the PL/I default file constants SYSIN and SYSPRINT. For example, to redirect output for the default file SYSPRINT, you could specify a command as follows:

```
$ DEFINE SYSPRINT TEST.OUT
```

While this assignment is in effect, any PL/I procedure that outputs data to SYSPRINT (without opening SYSPRINT with an explicit title) will create a file named TEST.OUT on the current default device.

Logical names may also be established by other commands. For example, you can specify a logical name for a device when you enter an ALLOCATE or MOUNT command while placing the device on line. For example:

```
$ ALLOCATE
$_Device:   MT:
$_Log_Name: INFILE
  _MTA1: ALLOCATED
```

This ALLOCATE command allocates a tape drive and establishes the logical name INFILE for it. When a PL/I program reads from the file INFILE, the system will translate the name INFILE and use the tape MTA1: as the input device.

System logical name table	PAY_DEV	DBB2:[1]
Group logical name table	WEEKLY_UPDATE	PAY_DEV:[WEEKLY.BACKUP]WEEK42.DAT[2]
Process logical name table	OUTFILE	WEEKLY_UPDATE[3]

1. This assignment may have been made when a disk volume was placed on line with a command, as follows:

```
$ MOUNT/SYSTEM DBB2:
$ _Label:        PAYROLL_FILES
$ _Log_Name:  PAY_DEV
```

2. This assignment may have been made as follows:

```
$ DEFINE/GROUP WEEKLY_UPDATE PAY_DEV:[WEEKLY.BACKUP]WEEK42.DAT
```

3. This assignment may have been made as follows:

```
$ DEFINE OUTFILE WEEKLY_UPDATE
```

```
OPEN FILE(OUTFILE) RECORD OUTPUT;
```

To determine the file specification, the system:

1. *Translates the name OUTFILE. The result is:*
   WEEKLY_UPDATE
2. *Translates the name WEEKLY_UPDATE. The result is:*
   PAY_DEV:[WEEKLY.BACKUP]WEEK42.DAT
3. *Translates the name PAY_DEV. The final resulting file specification is:*
   DBB2:[WEEKLY.BACKUP]WEEK42.DAT

Figure 5–1: Translating Logical Names

5.2.2.2 Process Permanent Logical Names — The system provides every user and every batch job with a default set of process logical name table assignments. These logical names are listed in Table 5–1. Because the files associated with these assignments exist for the life of the process, or job, and because they are permanently open, they are called process permanent files.

- The equivalence name for the logical name REPORT does not contain a file type. In this case, the file type LIS will be supplied by default to the translated equivalence of the logical name REPORT.

VAX-11 PL/I uses the punctuation in the DEFAULT_FILE_NAME option to determine which portion of the file specification is specified. Thus, the period (.) in the above example indicates that the value is a file type. An unpunctuated name is treated as a file name; a name terminated by a colon (:) is treated as a device name (and can therefore be a logical name).

When the DEFAULT_FILE_NAME option is not specified for a file, PL/I supplies a default value for the option of ".DAT"; that is, PL/I applies the file type DAT to a file specification that does not have a file type.

PL/I applies the value of the DEFAULT_FILE_NAME option after it establishes the file's title. Thus, in the preceding example, the title, REPORT, is established before the value ".LIS" is applied. Note that the only time that a file name in a DEFAULT_FILE_NAME option is used is when the TITLE option specifies a null string; that is, the TITLE option is specified as:

```
TITLE('')
```

A DEFAULT_FILE_NAME option can specify any portion of a file specification. For example:

```
DECLARE REMOTE_FILE FILE RECORD INPUT
        ENV(DEFAULT_FILE_NAME(
          'RONDO::DBB2:[MALCOLM].TXT'));
```

This option specifies a node name, device, directory, and file type. The file name must be supplied when the file is opened. For example:

```
OPEN FILE(REMOTE_FILE) TITLE('ALLEGRO');
```

This OPEN statement opens the file:

```
RONDO::DBB2:[MALCOLM]ALLEGRO.TXT
```

Another OPEN statement for the file may specify a different TITLE option, for example, TITLE('ANDANTE'), to open a different file. Note that if no TITLE option is specified in this example, the UNDEFINEDFILE condition will be signaled because the default title, REMOTE_FILE, is an invalid VAX/VMS file name.

5.2.4 Expanding File Specifications

After logical name translation and after values supplied by the DEFAULT_FILE_NAME option, if any, are applied, the defaults that the file system applies are as follows:

Field	System Default Provided
node	Local system
device	Current default device
directory	Current default directory
file name	None
file type	DAT
version number	For an input file, the most recent version; for an output file, the highest existing version number, plus 1.

Example:

```
DCL STATES FILE RECORD OUTPUT;
OPEN FILE (STATES);
```

Steps:

1. Apply the default title, STATES.
2. Translate the logical name STATES to obtain the equivalence name, DMA2:[BACKUP].
3. Apply default file type DAT and the default version number (for an output file). Note that no default is supplied for the file name.

Final Specification: DMA2:[BACKUP].DAT;*n* where *n* is one higher than the number of any existing version of the file

Example:

```
DCL TAPEFILE FILE RECORD ENVIRONMENT(
          DEFAULT_FILE_NAME('TAPEFILE:'));
OPEN FILE(TAPEFILE) OUTPUT TITLE('TAPE1.FIL');
```

Steps:

1. Apply the title TAPE1.FIL.
2. Translate the name TAPEFILE in the DEFAULT_FILE_NAME option to its equivalence, MTA0:.
3. Use the system default version number for tape files, 0. Tape files do not have directories.

Final Specification: MTA0:TAPE1.FIL;0

5.3 Error Handling

VAX-11 PL/I uses the standard PL/I ON condition names to signal run-time errors that occur for file operations. The ON conditions that are signaled, and the circumstances under which they are signaled, are:

- The UNDEFINEDFILE condition is signaled whenever a file cannot be opened.
- The ENDFILE condition is signaled when the end-of-file is reached during an input operation.
- The ENDPAGE condition is signaled for a file with the PRINT attribute when the current line number exceeds the page size specified for the file.
- The KEY condition is signaled for a file with the KEYED attribute when any error involving the interpretation, writing, or specification of a key occurs.
- The ERROR condition is signaled for all other file-related errors.

5.3.2 Default Error Handling

If a file system error occurs during the execution of a PL/I statement, the PL/I run-time system signals either the specific PL/I condition name or the ERROR condition. If no user-specified ON-units exist to handle either the specific PL/I condition or the ERROR condition, PL/I performs its default condition handling.

If any active procedure specified OPTIONS(MAIN), a default PL/I ON-unit is present and executed. The default PL/I ON-unit prints a PL/I run-time error message. If the default PL/I ON-unit is not present, the error signal is passed to the default condition handler established by VAX/VMS, which prints the message associated with the RMS error. If the error was a fatal error, the handler terminates the program; otherwise, the program continues.

The following example illustrates the type of messages that the PL/I run-time system displays when an error occurs during an I/O operation:

```
%PLI-F-ERROR, PL/I ERROR condition.
-PLI-I-IOERROR, I/O error on file 'STATE_FILE'
-PLI-I-FILENAME, File name: '_DB7:[MALCOLM]STDATA.DAT;'
-PLI-I-NOTKEYD, Not a KEYED file.
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name     line    relative PC     absolute PC

PLI$CONDIT      $CODE                    000004D8        000017A0
PLI$READ        $CODE                    00000336        0000294E
FLOWERS         BEGIN%35          35     00000085        00000C88
FLOWERS         BEGIN%29          29     000000BD        00000C02
FLOWERS         FLOWERS           25     000000D3        00000B42
```

In this example, the error occurred because a keyed I/O statement was specified for a file that does not have the KEYED attribute. For an explanation of the information in a traceback message, see Section 4.1.2, "Run-Time Errors."

5.4 Input/Output Optimization

Many of the VAX-11 PL/I options for the ENVIRONMENT attribute provide optimization features for input/output operations. Table 5-2 summarizes the options that control disk file allocation. These options let you specify the space requirements of a file when you create it. Table 5-3 summarizes the options for run-time optimization of input/output processing. All options are described in detail in Chapter 6, "ENVIRONMENT Options."

Chapter 6
Options of the ENVIRONMENT Attribute

The options to the ENVIRONMENT attribute provided by VAX-11 PL/I let you:

- Describe the attributes of a file when it is created
- Request special processing and optimization options when the file is being read or written
- Specify the disposition of a file when it is closed

Most of the options for the ENVIRONMENT attribute correspond directly to RMS options and control values. PL/I, in some cases, provides different defaults than does RMS.

This chapter presents an overview of the ENVIRONMENT options and information on how to specify them and gives a reference description of each option. The descriptions of the ENVIRONMENT options begin in Section 6.2.1, and are arranged in alphabetic order.

6.1 Specifying and Using ENVIRONMENT Options

All ENVIRONMENT options may be specified in the declaration of a file constant or in an OPEN statement to open a file. Certain options may also be specified in a CLOSE statement.

6.1.1 Arguments for ENVIRONMENT Options

ENVIRONMENT options may be grouped in the following categories, based on whether they require an argument and what type of argument is required:

- Many options require you to specify an expression representing a value to override a default value provided by VAX-11 PL/I.
- A few ENVIRONMENT options require you to provide a reference to a variable that either contains information pertaining to the open or that will receive information when the related file is opened.
- All options that are not in one of the above categories may be specified with a Boolean expression that enables or disables the option. If no value is specified with an option, the option is enabled.

6.1.2 Interpretation of ENVIRONMENT Options for Existing Files

Many ENVIRONMENT options specify values that can be set only when a file is created. For example, the length of records in a file with fixed-length records is set when the file is created and cannot be changed thereafter. When these options are specified for a file, they are applied to the file only if the open of the file actually results in the creation of a new file. If the open results in the opening of a file that already exists, the option is ignored.

6.1.3 Determining ENVIRONMENT Options

A PL/I program can determine the value or setting of an ENVIRONMENT option at run time for an indicated file by calling the DISPLAY built-in subroutine. This built-in subroutine returns information about a specified PL/I file to a user-specified structure. The member names in the structure correspond to the keywords of the ENVIRONMENT attribute.

For a description of the values returned by this subroutine, and an example of calling it, see Section 8.1, "DISPLAY Built-In Subroutine."

Certain ENVIRONMENT options themselves return information to the program when an existing file is opened. For example, the FIXED_CONTROL_SIZE_TO option may be specified when an existing file with a fixed control area is opened. PL/I returns the size of the fixed control area to the program.

6.1.4 Device Independence of ENVIRONMENT Options

Many ENVIRONMENT options apply only to a particular type of device or to a specific file organization. For example, the REWIND_ON_CLOSE and REWIND_ON_OPEN options apply only to magnetic tape files, and the FILE_SIZE option applies only to disk files.

When any ENVIRONMENT option is specified for a device to which the option does not apply, the option is ignored.

6.1.5 Conflicting and Invalid ENVIRONMENT Options

Conflicting or invalid options or values for options may be detected during compilation or at run time. At compile time, the compiler issues a diagnostic message to indicate the error.

At run time, the UNDEFINEDFILE condition is signaled if conflicting options are in effect or if conflicting values are specified for the same option. For example, if the FILE_SIZE option is specified in the DECLARE and OPEN statements for a given file and if the options specify different values, UNDEFINEDFILE is signaled.

For run-time errors, an ON-unit can reference the ONCODE built-in function to determine the specific error, if desired. If no ON-unit exists for the UNDEFINEDFILE condition, the PL/I run-time system displays an error message describing the error that occurred.

Table 6-1: Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
APPEND	Places output for a file at the end of an existing file.	Create Open	Record Stream	Disabled	BIT(1)
BATCH	Submits a copy of the file to the system batch job queue on close.	Create Open Close	Record Stream	Disabled	BIT(1)
BLOCK_BOUNDARY_FORMAT	Indicates that records must not cross block boundaries.	Create	Record Stream	Disabled	BIT(1)
BLOCK_IO	Specifies a file will be read or written by blocks instead of records.	Create Open	Record	Disabled	BIT(1)
BLOCK_SIZE(expression)	Specifies the size of a block for the creation of a magnetic tape file.	Create	Record Stream	Mount value	FIXED BINARY(31)
BUCKET_SIZE(expression)	Defines the number of 512-byte blocks in a bucket for an indexed sequential or a relative file.	Create	Record	Maximum record size	FIXED BINARY(31)
CARRIAGE_RETURN_FORMAT	Indicates that records in the file will be printed with default carriage control.	Create	Record	Enabled	BIT(1)
CONTIGUOUS	Specifies that an output file must be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CONTIGUOUS_BEST_TRY	Requests that if possible an output file be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CREATION_DATE(variable)	Overrides default creation date of file.	Create	Record Stream	Current date and time	BIT (64) ALIGNED
CURRENT_POSITION	Leaves magnetic tape positioned at last close.	Create Open	Record Stream	Disabled	BIT(1)

(Continued on next page)

Table 6-1(Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
INDEX_NUMBER(expression)	Specifies the initial index to use in accessing records in an indexed sequential file.	Create Open	Record	0	FIXED BINARY(31)
INDEXED	Defines an indexed sequential file.	Create Open	Record	Disabled	BIT(1)
INITIAL_FILL	Requests the file system to leave unused space in file index overflow buckets.	Open	Record	Disabled	BIT(1)
MAXIMUM_RECORD_NUMBER(expression)	Specifies the largest record number that will be valid for records in a relative file.	Create	Record	0	FIXED BINARY(31)
MAXIMUM_RECORD_SIZE(expression)	Specifies the maximum size that is valid for any record in the file.	Create	Record	512 bytes*	FIXED BINARY(31)
MULTIBLOCK_COUNT(expression)	Specifies the number of blocks to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
MULTIBUFFER_COUNT(expression)	Specifies the number of buffers to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
NO_SHARE	Prohibits all type of shared access to the file.	Create Open	Record	†	BIT(1)
OWNER_GROUP(expression)	Specifies the group number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process group number	FIXED BINARY(31)

*For sequential files with fixed-length records. For sequential files with variable-length records, the default is 510 bytes. For relative files, the default is 480 bytes.

† Disabled if the file is opened for input, enabled if opened for output or update.

(Continued on next page)

Table 6-1(Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
SCALARVARYING	Specifies that varying character strings will be read/written using the entire storage of the variable.	Create Open	Record	Disabled	BIT(1)
SHARED_READ	Allows other users to read records in the file.	Create Open	Record	*	BIT(1)
SHARED_WRITE	Allows other users to read and write records in the file.	Create Open	Record	Disabled	BIT(1)
SPOOL	Queues a copy of the file to the system printer when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
SUPERSEDE	Replaces an existing file with the same file name, file type, and version number.	Create	Record Stream	Disabled	BIT(1)
SYSTEM_PROTECTION(expression)	Defines the type of file access allowed to users with system user identification codes.	Create	Record Stream	Current process default	CHAR(4)
TEMPORARY	Specifies a temporary file for which no directory entry is made.	Create	Record Stream	Disabled	BIT(1)
TRUNCATE	Truncates a sequential file at its logical end-of-file when it is closed.	Create Update Close	Record Stream	Disabled	BIT(1)
WORLD_PROTECTION(expression)	Specifies the type of file access allowed to general system users.	Create	Record Stream	Current process default	CHAR(4)
WRITE_BEHIND	Requests file system optimization on output operations.	Create Update	Record Stream	Disabled	BIT(1)
WRITE_CHECK	Requests verification of output operations.	Create Update	Record Stream	Disabled	BIT(1)

* Enabled if the file is opened for input, otherwise disabled.

■ Usage

When you specify both the TEMPORARY and DELETE options in conjunction with the BATCH option, the file is submitted to the batch job queue and marked for deletion after the batch job completes.

6.2.3 BLOCK_BOUNDARY_FORMAT Option

The BLOCK_BOUNDARY_FORMAT option indicates that records in the file must not cross block boundaries.

The format of this option is:

```
BLOCK_BOUNDARY_FORMAT [ (boolean-expression) ]
```

■ Rules

1. The BLOCK_BOUNDARY_FORMAT option is meaningful only when a file is created.
2. This option applies only to sequential files; it is ignored if specified for relative or indexed sequential files.
3. If the BLOCK_BOUNDARY_FORMAT option is specified for a file, the maximum record size must be less than 512 bytes.
4. BLOCK_BOUNDARY_FORMAT conflicts with the BLOCK_IO option. However, a file that is created with the BLOCK_BOUNDARY_FORMAT option can later be read with the BLOCK_IO option.

■ Usage

The BLOCK_BOUNDARY_FORMAT option can be paired with the CARRIAGE_RETURN_FORMAT or PRINTER_FORMAT option to define the attributes of a file's records.

This option may be useful for the creation of files that will be read in terms of blocks. Note, however, that this option may result in unused disk space when records do not fill blocks.

6.2.4 BLOCK_IO Option

The BLOCK_IO option indicates that all I/O operations on the file will be in terms of physical blocks rather than records. In an I/O statement, a block is treated as if it were a single logical record. The format of this option is:

```
BLOCK_IO [ (boolean-expression) ]
```

■ Rules

1. The BLOCK_IO option is meaningful when a file is created or opened. The file can be opened with any of the attributes INPUT, OUTPUT, or UPDATE. If the file is opened for output, the created file is always sequential.

■ Rules

1. The BLOCK_SIZE option is meaningful only when a file is created.
2. This option is applied only to magnetic tape files.

■ Usage

When a tape file is opened with the BLOCK_IO option of ENVIRONMENT, the block size of the file is used to determine the number of bytes to be transferred in a single I/O operation.

Tape file input/output is described in Section 10.2, "Using Magnetic Tape Files."

6.2.6 BUCKET_SIZE Option

The BUCKET_SIZE option lets you specify the number of blocks to use for each bucket when you create a relative file. The BUCKET_SIZE option has the format:

BUCKET_SIZE(integer-expression)

integer-expression
Is a fixed binary value in the range of 0 to 32, representing the number of blocks in each bucket. If the bucket size is specified as 0, or if it is not specified, PL/I applies the current RMS default. This default can be set with the DCL command SET RMS_DEFAULT; its current value can be determined with the command SHOW RMS_DEFAULT.

■ Rules

1. The BUCKET_SIZE option is meaningful only when a file is created.
2. This option applies only to relative files.

■ Usage

Selection of a bucket size for a relative file depends on the size of the records in the file. Although records within a bucket can cross block boundaries, records cannot cross bucket boundaries. Therefore, the number of blocks per bucket that you specify with this option must conform to one of the formulas given below.

By careful calculation of a bucket size, you can improve input/output operations on the file. In general, a bucket size of between four and eight blocks results in good performance for most files. For detailed information on file design and space considerations, see the *RMS-11 User's Guide*.

6.2.7 CARRIAGE_RETURN_FORMAT Option

The CARRIAGE_RETURN_FORMAT option indicates that each record in the file is to be preceded by a line feed and followed by a carriage return when the line is written to a carriage control device such as a terminal or line printer. The format of this option is:

CARRIAGE_RETURN_FORMAT [(boolean-expression)]

■ Rules

1. The CARRIAGE_RETURN_FORMAT option is meaningful only when a file is created.
2. CARRIAGE_RETURN conflicts with the PRINTER_FORMAT and BLOCK_IO options and with the PRINT file description attribute.

■ Usage

CARRIAGE_RETURN_FORMAT is the default format for record files.

This type of carriage control is an attribute of the file that is known to the file system; it does not require space within the file's records.

6.2.8 CONTIGUOUS Option

The CONTIGUOUS option specifies that disk space for the associated file must be allocated using contiguous blocks on the disk. The format of this option is:

CONTIGUOUS [(boolean-expression)]

■ Rules

1. The CONTIGUOUS option is meaningful only when a file is created.
2. This option applies only to disk files.
3. If specified with the CONTIGUOUS_BEST_TRY option, the CONTIGUOUS_BEST_TRY option takes precedence.

■ Usage

By default, a disk file consists of areas, or extents, on a disk volume that are not contiguous. When a file is accessed, the file system must maintain a pointer to each extent. However, there is a maximum number of extents that can be maintained. For very large files that must be accessed quickly, an initial allocation of contiguous space can result in more efficient input/output operations.

■ Usage

The time value required can be obtained by using the SYS$BINTIM (Convert ASCII String to Binary Time) system service procedure. For an example of a call to this procedure to obtain a system time value for the CREATION_DATE option, see Section 19.4.3, "Timer and Time Conversion Routines."

6.2.11 CURRENT_POSITION Option

The CURRENT_POSITION option specifies that a magnetic tape volume be positioned immediately after the most recently closed file when the next file is created. The format of this option is:

CURRENT_POSITION [(boolean-expression)]

■ Rules

1. The CURRENT_POSITION option is meaningful only when a file is created.
2. This option applies only to magnetic tape files.
3. If the REWIND_ON_OPEN option is also selected, it takes precedence over the CURRENT_POSITION option.

■ Usage

This option lets you close an output file on a magnetic tape and proceed to write another file on the tape immediately after the current file. For example:

```
DECLARE TAPEFILE FILE RECORD OUTPUT ENV(
          DEFAULT_FILE_NAME('TAPEFILE:'));
OPEN FILE(TAPEFILE) ENV(CURRENT_POSITION)
            TITLE('TAPE1.FIL');
   .
   .
CLOSE FILE(TAPEFILE);
OPEN FILE(TAPEFILE) TITLE('TAPE2.FIL');
```

When the second OPEN statement executes, the tape identified by the logical name TAPEFILE remains positioned as it was following the CLOSE statement.

Magnetic tape file positioning is described in Section 10.2, "Using Magnetic Tape Files."

■ Usage

The DEFERRED_WRITE option can provide better I/O performance for output operations, especially when a relative or indexed sequential file is being initially loaded with records and the records are being added sequentially.

If a system problem occurs when I/O is being performed with the DEFERRED_WRITE option enabled, data may be lost. To ensure the integrity of the file during processing with this option, a PL/I program can call the FLUSH built-in subroutine at critical times to rewrite all buffers. The FLUSH built-in subroutine is described in Chapter 8, "File-Handling Built-In Subroutines."

6.2.14 DELETE Option

The DELETE option specifies that the file is to be deleted when it is closed. The format of this option is:

DELETE [(boolean-expression)]

■ Rules

1. The DELETE option can be specified when a file is created, opened, or closed.
2. Once the DELETE option has been enabled for a file on a particular open, it cannot be disabled.

■ Usage

When this option is used in conjunction with the SPOOL or BATCH options, the file is marked to be deleted after it is either printed or processed as a batch job.

This option can also be used to delete an existing file. For example:

```
DECLARE INFILE FILE;
OPEN FILE (INFILE) ENVIRONMENT(DELETE);
CLOSE FILE(INFILE);
```

When this CLOSE statement executes, the VAX/VMS file associated with the PL/I file constant INFILE is deleted.

6.2.15 EXPIRATION_DATE Option

The EXPIRATION_DATE option specifies the time at which a magnetic tape file expires. The file cannot be deleted or overwritten until the specified date. The format of the EXPIRATION_DATE option is:

EXPIRATION_DATE (variable-reference)

Each time the addition of a record to a file requires the file system to allocate additional disk extents for the file, RMS allocates the amount of space specified by the EXTENSION_SIZE value. Thus, if you specify a value that is larger than the default that RMS uses, the number of times that a file must be extended will be decreased.

However, if a large extension quantity is specified for a file, and the file does not require the allocated space, the disk space is wasted.

6.2.17 FILE_ID Option

When the FILE_ID option is specified in the opening of an existing file, PL/I uses the value specified in the FILE_ID option to locate the file. This format of the option is:

FILE_ID(variable-reference)

variable-reference
Specifies the name of a six-element array variable that gives the file identification obtained when the file was created.

The variable must be declared as (6) FIXED BINARY(31) and must be connected.

■ Rules

1. The FILE_ID option is valid only when an existing file is opened.
2. This option conflicts with the TITLE, DEFAULT_FILE_NAME, and FILE_ID_TO options.
3. If there is no file with the indicated file identification, the UNDEFINED-FILE condition is signaled.

■ Usage

This option is provided for use with the TEMPORARY option; you must specify the FILE_ID option to reopen a file that was created with the TEMPORARY option.

6.2.18 FILE_ID_TO Option

When a file is created, the FILE_ID_TO option requests PL/I to return the file identification to a user-specified variable. Its format is:

FILE_ID_TO(variable-reference)

variable-reference
Specifies the name of a six-element array variable to receive the file identification of the created file.

The variable must be declared as (6) FIXED BINARY(31) and must be connected.

If the specified file size is not a multiple of the cluster size of the disk, the allocation is rounded up to a multiple of the cluster size.

Note that if you allocate more space for a file than it requires, the unused space is wasted, since it is unavailable for other uses.

6.2.20 FIXED_CONTROL_SIZE Option

The FIXED_CONTROL_SIZE option specifies that a file will have a fixed-length control area associated with each variable-length record and specifies the size of the fixed control area.

The format of this option is:

FIXED_CONTROL_SIZE(integer-expression)

integer-expression
Is an integer expression in the range 0 to 255, indicating the number of bytes in the fixed control field of the record. If you specify a value of 0, PL/I uses the default size of two bytes.

■ Rules

1. The FIXED_CONTROL_SIZE option is meaningful only when a file is created.
2. This option applies only to relative and sequential files with variable-length records.
3. The FIXED_CONTROL_SIZE option conflicts with the BLOCK_IO and INDEXED options and with the STREAM and UPDATE file description attributes.
4. You must specify the FIXED_CONTROL_SIZE option to create a file containing records with a fixed-length control area.

■ Usage

When a file is created with the FIXED_CONTROL_SIZE option, WRITE and REWRITE statements for the file may specify the FIXED_CONTROL_FROM option to write a value into the fixed control area. For example:

```
DECLARE OUTFILE FILE RECORD OUTPUT ENVIRONMENT (
                        FIXED_CONTROL_SIZE (2));

OPEN FILE(OUTFILE);
WRITE FILE (OUTFILE) FROM (NEWLINE) OPTIONS (
              FIXED_CONTROL_FROM(LINE_NUMBER));
```

If the FIXED_CONTROL_FROM option is not specified when a record is written to a file with fixed control records, VAX-11 PL/I writes zeros in the fixed control area of the record.

The format of variable-length records with a fixed-length control area is described in Section 9.3.3, "Variable-Length Records with a Fixed-Length Control Area." For an additional example of writing a file with a fixed control area, see Section 7.2.3, "FIXED_CONTROL_FROM Option."

6.2.23 GROUP_PROTECTION Option

The GROUP_PROTECTION option defines the type of access to be permitted to the file by other users in the owner's group. The format of this option is:

GROUP_PROTECTION (character-expression)

character-expression
Is a one- to four-character string expression indicating the access privileges to be granted to users in the owner's group. The expression can contain any of the following letters to indicate the access allowed:

Letter	Meaning
R	Read access is allowed
W	Write access is allowed
E	Execute access is allowed
D	Delete access is allowed

The lowercase forms of these letters are also permitted. Letters may be repeated, but the maximum length of the string is four. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

■ Rules

1. The GROUP_PROTECTION option is meaningful only when a file is created.
2. If no protection options are specified, PL/I uses the current system and process defaults. If any protection options are specified, the protection for unspecified user categories defaults to no access.

■ Usage

For information on specifying protection options, see Chapter 13, "File Protection and File Sharing."

6.2.24 IGNORE_LINE_MARKS Option

The IGNORE_LINE_MARKS option overrides the default manner in which VAX-11 PL/I interprets end-of-line indicators on stream input operations, which is to treat an end-of-line on a stream input operation as a field delimiter in a GET LIST or GET EDIT statement. The format of this option is:

IGNORE_LINE_MARKS [(boolean-expression)]

■ Rules

1. The IGNORE_LINE_MARKS option may be specified when a file is opened.
2. This option applies only to stream input files; that is, it conflicts with the RECORD, OUTPUT, and UPDATE attributes and with any attributes that imply these attributes.

6.2.26 INDEXED Option

The INDEXED option specifies that a file is an indexed sequential file. The format of this option is:

INDEXED [(boolean-expression)]

■ Rules

1. The INDEXED option is meaningful when an existing file is opened.
2. This option applies only to indexed sequential files.
3. INDEXED conflicts with the APPEND, BATCH, BLOCK_IO, FIXED_CONTROL_SIZE, MAXIMUM_RECORD_NUMBER, and PRINTER_FORMAT options and with the PRINT file description attribute.

■ Usage

The INDEXED option is never required; however, you may use it as a check when you open an existing indexed sequential file so that PL/I will verify the file's organization before opening it.

6.2.27 INITIAL_FILL Option

The INITIAL_FILL option specifies, when an indexed sequential file is opened, that the initial fill value specified when the file was created is to be used. The format of this option is:

INITIAL_FILL [(boolean-expression)]

■ Rules

The INITIAL_FILL option is meaningful only when an indexed sequential file is initially opened for output.

■ Usage

As an indexed sequential file is initially loaded with records, the fill size specified causes buckets to appear full when they are actually less than full. Thus, room remains in each bucket for subsequent additions to the file.

For information on using indexed sequential files, see Chapter 12, "Indexed Sequential Files."

6.2.28 MAXIMUM_RECORD_NUMBER Option

The MAXIMUM_RECORD_NUMBER option sets, for a relative file, the largest record number that can be written to the file.

The format of this option is:

MAXIMUM_RECORD_NUMBER(integer-expression)

integer-expression

Is a numeric expression with values in the range of 1 to a maximum determined by record format and file organization, as follows:

File Organization	Record Format	Maximum Allowed
Sequential	Fixed or variable length	32,767
Relative	Fixed length	16,383
Relative	Variable length	16,381
Indexed sequential	Fixed length	16,362
Indexed sequential	Variable length	16,360

For variable-length records with a fixed-length control area, the size of the fixed control area must be subtracted from the maximum value allowed.

A value of 0 indicates that there is no user-defined limit to the size of records.

If the value is out of range, the UNDEFINEDFILE condition is signaled.

■ Rules

The MAXIMUM_RECORD_SIZE option is meaningful only when a file is created. If not specified, PL/I provides a default length based on the file organization and record format, as follows:

File Organization	Record Format	Default
Sequential	Fixed length	512
Sequential	Variable length	510
Relative	Fixed or variable length	480

If the file has variable with fixed-length control records, the size of the fixed control area is subtracted from the default value listed above.

6.2.30 MULTIBLOCK_COUNT Option

The MULTIBLOCK_COUNT option specifies the number of blocks to allocate in each internal buffer for operations on a sequential disk file. Its format is:

MULTIBLOCK_COUNT(integer-expression)

integer-expression

Is a fixed binary expression in the range of 0 to 127, indicating the number of blocks to be allocated to each buffer. If 0 is specified, PL/I uses the system default. You can determine the current system default by entering the DCL command SHOW RMS_DEFAULT. Use the SET RMS_DEFAULT command to establish a new default value, if desired.

If the value is not within the required range, the UNDEFINEDFILE condition is signaled.

■ Rules

1. The MULTIBUFFER_COUNT option is meaningful when a file is created or opened.
2. This option applies only to disk files.
3. This option has no effect if BLOCK_IO is specified.

■ Usage

When you use the MULTIBUFFER_COUNT option, it decreases the number of actual data transfers and thus increases a program's execution speed. For example:

```
OPEN FILE(REL_FILE)
     ENVIRONMENT(
                 READ_AHEAD,
                 MULTIBLOCK_COUNT(4),
                 MULTIBUFFER_COUNT(4));
```

This option can be specified for sequential, relative, or indexed sequential files. For inserting records in an indexed sequential file, a good rule of thumb is to specify one buffer for each index in use, plus two or more buffers for data. Thus, an indexed sequential file with a primary key and two alternate keys could be opened with:

```
ENVIRONMENT(MULTIBUFFER_COUNT(5))
```

This option specifies five buffers.

Multibuffering is also effective for sequential files when combined with the ENVIRONMENT options READ_AHEAD or WRITE_BEHIND. These options are described individually in this chapter.

6.2.32 NO_SHARE Option

The NO_SHARE option prohibits sharing of the data in a file. The format of the NO_SHARE option is:

NO_SHARE [(boolean-expression)]

■ Usage

Note that although the value may be specified to PL/I in decimal, the VAX/VMS system always displays UICs in octal format. For information on specifying protection options, see Chapter 13, "File Protection and File Sharing."

6.2.34 OWNER_MEMBER Option

The OWNER_MEMBER option overrides the default member number in the user identification code associated with the file's owner. The member number of a file's owner, together with the group number, provides protection for the file. Its format is:

OWNER_MEMBER(integer-expression)

integer-expression
Is a numeric value in the range of 0 to 255.

■ Rules

1. The OWNER_MEMBER option is meaningful only when a file is created.
2. If not specified, PL/I uses the member number in the current UIC.
3. To specify an owner UIC for a file that is different than the UIC under which the current program is executing, the process must have the SYSPRV user privilege or must have a system UIC.

■ Usage

Note that although the value may be specified to PL/I in decimal, the VAX/VMS system always displays UICs in octal format. For information on specifying protection options, see Chapter 13, "File Protection and File Sharing."

■ Rules

1. The PRINTER_FORMAT option is meaningful only when a file is created.
2. The FIXED_CONTROL_SIZE option should be specified with the PRINTER_FORMAT option. The size of the fixed control area must be two to six bytes. If FIXED_CONTROL_SIZE is not specified, the size of the fixed control area defaults to two bytes.
3. This option may be applied only to relative or sequential files
4. PRINTER_FORMAT conflicts with the STREAM file description attribute and with the following ENVIRONMENT options:

   CARRIAGE_RETURN_FORMAT
   FIXED_LENGTH_RECORDS
   BLOCK_IO

■ Usage

This option indicates that a file is in printer format, that is, that the fixed control area of each record contains carriage control information. Printer file format provides more explicit carriage control than the default type of carriage control, called carriage return format. Printer format is particularly useful in formatting a printed listing.

Table 6–2 summarizes the coding specifications for the fixed-length control area for files with printer format. The first byte in the fixed control area is called the prefix byte: it gives the carriage control to perform before writing the record. The second byte is the postfix byte: it gives the carriage control to perform after writing the record. The values shown in Table 6–2 have the same meanings in either byte; the bytes are interpreted separately.

Table 6–2: Printer File Format Carriage Control

Bit 7	Bits 0 - 6	Meaning
0	0	No carriage control
0	1 - '7F'B4	Bits 0 through 6 contain the count of line feeds

Bit 7	Bit 6	Bit 5	Bits 0 - 4	Meaning
1	0	0	1 - '1F'B4	The carriage control is specified by the ASCII value in bits 0 through 4.

The carriage controls associated with the ASCII values are listed in the table of ASCII codes in Appendix D.

Sample Program 6–1: Explicit Carriage Control

```
PRINTER_FORMAT_EXAMPLE: PROCEDURE OPTIONS(MAIN);/1

/* Declare structure definitions for carriage control bit fields */
/* and a FIXED BIN(15) variable for the fixed control area       */ ❶

DECLARE 1 LINE_FEEDS STATIC,
          2 COUNT BIT(7),       /* contains count of line feeds */
          2 INDICATOR BIT(1) INIT('0'B),/* must be zero */
        1 CARRIAGE_CONTROL STATIC,
          2 CODE BIT(5),        /* bits 0-4 ASCII code for action */
          2 FILLER BIT(2) INIT('00'b),          /* bits 5 and 6 */
          2 EXPLICIT BIT(1) INIT('1'B),         /* bit 7 must be set */
        CONTROL_FIELD BIT(16) ALIGNED;

/* Set up variables for Form Feeds and CRs */

DECLARE (NEW_LINE,NEW_PAGE) BIT(8),
        I FIXED;                                ❷

        I = 12;             /* ASCII decimal code for Form Feed */
        CODE = UNSPEC(I);       /* assign 12 to CODE field */
        NEW_PAGE = STRING(CARRIAGE_CONTROL);

        I = 13;             /* ASCII decimal code for CR */
        CODE = UNSPEC(I);       /* assign 13 to CODE field */
        NEW_LINE = STRING(CARRIAGE_CONTROL);

/* declare and open PRINTFILE, with character-string variable for I/O */
DECLARE PRINTFILE RECORD OUTPUT FILE ENV(
                FIXED_CONTROL_SIZE(2),      ❸
                PRINTER_FORMAT),
        PRINTREC CHARACTER(80) VARYING;

OPEN FILE(PRINTFILE);

/* output first line with no carriage control */
        PRINTREC = 'Output first line with no carriage control'; ❹
        WRITE FILE(PRINTFILE) FROM(PRINTREC);

/* Prepare to output five line feeds followed by a new line */

        I = 5;                  /* assign 5 to LINE__FEEDS.COUNT */
        LINE_FEEDS.COUNT = UNSPEC(I);                            ❺
        CONTROL_FIELD = STRING(LINE_FEEDS)||NEW_LINE;
        PRINTREC = 'Record preceded by 5 line feeds ';

        WRITE FILE(PRINTFILE) FROM (PRINTREC) OPTIONS(
                FIXED_CONTROL_FROM(CONTROL_FIELD));

/* Prepare to output a page eject followed by a new line */

CONTROL_FIELD = NEW_PAGE||NEW_LINE;
        PRINTREC = 'New page';                   ❻

        WRITE FILE(PRINTFILE) FROM(PRINTREC) OPTIONS(
                FIXED_CONTROL_FROM(CONTROL_FIELD));   ❼

        CLOSE FILE(PRINTFILE) ENV(SPOOL);
        END;                                     ❽
```

6.2.39 RECORD_ID_ACCESS Option

The RECORD_ID_ACCESS option indicates that the records in a file will be accessed randomly using the internal identification of the records. The format of this option is:

RECORD_ID_ACCESS [(boolean-expression)]

■ Rules

1. The RECORD_ID_ACCESS option is meaningful when a file is created or opened.
2. This option applies only to disk files.
3. The RECORD_ID_ACCESS option conflicts with the BLOCK_IO option.

■ Usage

You must open a file with this option to use the RECORD_ID_TO and RECORD_ID options of the record I/O statements. These options are described in Chapter 7, "I/O Statement Options."

When a file is opened with the RECORD_ID_ACCESS option, access by record identification can be mixed with sequential access or access by key during this open of the file. However, a statement cannot specify a record both by key and by record identification.

6.2.40 RETRIEVAL_POINTERS Option

The RETRIEVAL_POINTERS option specifies the number of extent pointers to be maintained in main memory for file access. Each pointer provides access to a separate extent in the file; increasing the number of pointers for a noncontiguous file can increase the speed with which records are accessed during I/O operations. Its format is:

RETRIEVAL_POINTERS(integer-expression)

integer-expression
Is a fixed binary expression in the range of 0 to 127, or -1. A value in the range of 1 to 127 indicates the number of pointers. If you specify -1, the file system maps as much of the file as possible. If the option is not specified, or if the expression has a value of 0, the file system uses the default number established when the volume was initialized or mounted.

■ Rules

The RETRIEVAL_POINTERS option is meaningful when a file is created or opened.

■ Usage

Magnetic tape file positioning is described in Section 10.2, "Using Magnetic Tape Files."

6.2.43 SCALARVARYING Option

The SCALARVARYING option specifies that character strings with the VARYING attribute will be read and written in strict accordance with the PL/I ANSI standard. Its format is:

SCALARVARYING [(boolean-expression)]

■ Rules

1. The SCALARVARYING option is meaningful when a file is created or opened.
2. SCALARVARYING conflicts with the STREAM file description attribute.

■ Usage

The SCALARVARYING option has the following effect on input/output operations involving VARYING character-string variables:

- When a record is written from a varying-length character string, the entire storage of the string is written, including the word containing the string's current length.
- When a record is read into a varying-length character-string variable, the first word of the record is read into the variable's current length field.

Thus, records to be read into or from variables with the VARYING attribute should be images of a varying character string — including the two-byte count field at the beginning of the string.

When SCALARVARYING is not specified, character-string variables with the VARYING attribute are handled so as to facilitate reading and writing files with variable-length records. The rules are:

- On an input operation, the entire record read into the variable is treated as a character string and assigned to the variable. Thus, the current length of the variable is always set to the record length of the record read, unless truncation occurs.
- On an output operation, only the characters of the string's current value are written.

For strings with the VARYING attribute that are embedded in arrays or structures, the entire storage is always read or written.

When a file is to be read with SCALARVARYING in effect, the target variable must be declared CHARACTER VARYING and the length of the target variable must match the record length of each record in the file, minus two bytes. If the length does not match, the ERROR condition is signaled.

■ Rules

1. The SHARED_WRITE option is meaningful when a file is created or opened.
2. This option applies to relative and indexed sequential files.
3. SHARED_WRITE conflicts with the NO_SHARE option.
4. If SHARED_READ and SHARED_WRITE are both specified, the effect is the same as if only SHARED_WRITE were specified.

■ Usage

By default, the SHARED_WRITE option is disabled.

For information on file sharing, see Chapter 13, "File Protection and File Sharing."

6.2.46 SPOOL Option

The SPOOL option requests that the file be submitted to the system printer job queue when it is closed. The format of this option is:

SPOOL [(boolean-expression)]

■ Rules

1. The SPOOL option can be specified when a file is created, opened, or closed.
2. This option applies to stream files as well as to record files of any file organization.
3. Once the SPOOL option has been specified for a file on a particular open, it cannot be disabled.

■ Usage

If you specify the DELETE option in conjunction with the SPOOL option, the file is submitted to the queue SYS$PRINT when it is closed and marked to be deleted after printing.

You can control the queue to which the file is submitted by using the DEFINE command to equate the logical name SYS$PRINT with the name of a specific queue before running the program. For example:

```
$ DEFINE SYS$PRINT LPC0:
$ RUN PRINTER
```

If the PL/I program PRINTER closes a file with the SPOOL option, the file is queued to the printer device LPC0:.

The lowercase forms of these letters are also permitted. Letters may be repeated, but the maximum length of the string is four. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

■ Rules

1. The SYSTEM_PROTECTION option is meaningful only when a file is created.
2. If no protection options are specified, PL/I applies the current system and process defaults. If any protection options are specified, the protection for unspecified user categories defaults to no access.

■ Usage

For information on specifying protection options, see Chapter 13, "File Protection and File Sharing."

6.2.49 TEMPORARY Option

The TEMPORARY option creates a temporary file with no directory entry. The format of this option is:

TEMPORARY [(boolean-expression)]

■ Rules

1. The TEMPORARY option is meaningful only when a file is created.
2. TEMPORARY conflicts with the TITLE and the DEFAULT_FILE_NAME options.

■ Usage

When you create a file with the TEMPORARY option, the file system does not create a directory entry for the file. A file thus created can be used during the execution of the program and deleted on completion, without the overhead required to create and remove the directory entry.

The file may be deleted when it is closed or, if needed later, deleted after it has been reused.

■ Usage

You can specify this option to conserve disk space. If a file's allocation is greater than is required for the contents of the file, and if the file is not expected to increase in size, you may want to use this option to reclaim the allocated but unused space.

6.2.51 WORLD_PROTECTION Option

The WORLD_PROTECTION option defines the type of access to be permitted to the file by users who are not in the owner's group and who do not have system user identification codes. The format of this option is:

WORLD_PROTECTION(character-expression)

character-expression
Is a one- to four-character string expression indicating the access privileges to be granted to users in the world category. The character-string expression can contain any of the following letters to indicate the access allowed:

Letter	Meaning
R	Read access is allowed
W	Write access is allowed
E	Execute access is allowed
D	Delete access is allowed

The lowercase forms of these letters are also permitted. Letters may be repeated, but the maximum length of the string is four. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

■ Rules

1. The WORLD_PROTECTION option is meaningful only when a file is created.
2. If no protection options are specified, PL/I uses the current system and process defaults. If any protection options are specified, the protection for unspecified user categories defaults to no access.

■ Usage

For information on specifying protection options, see Chapter 13, "File Protection and File Sharing."

6.2.52 WRITE_BEHIND Option

The WRITE_BEHIND option requests the file system to overlap the writing of buffers with computing operations. The format of this option is:

WRITE_BEHIND [(boolean-expression)]

Chapter 7
I/O Statement Options

VAX-11 PL/I permits the specification of the OPTIONS keyword on I/O statements and supports certain options for each statement. This chapter describes how to code options for I/O statements, lists the valid options for each I/O statement, and describes each option individually.

7.1 How to Code I/O Statement Options

All options are specified in an option list following the OPTIONS keyword, as follows:

OPTIONS (option,...) ;

Options must be separated by commas and enclosed in parentheses. For example:

```
GET LIST (PASSWORD) OPTIONS (
                             PROMPT('Enter password: '),
                             NO_ECHO,
                             PURGE_TYPE_AHEAD);
```

Any option that does not require an argument may be followed by a Boolean expression in the format:

option(boolean-expression)

If no Boolean expression is specified and the option is present in the option list, the default value of true is supplied.

7.2 Summary of I/O Statement Options

Table 7-1 lists the I/O options and indicates which options are valid for each I/O statement.

7.2.1 CANCEL_CONTROL_O Option

The CANCEL_CONTROL_O option specifies, when the output device is a terminal, that the effect of CTRL/O is disabled prior to output. This ensures that the beginning of the output list is displayed.

■ Rules

1. The CANCEL_CONTROL_O option is valid only on a PUT statement.
2. This option is ignored when the output device is any device other than an interactive terminal.

■ Usage

Use this option on a PUT statement that you want to display regardless of whether previous output has been interrupted by CTRL/O. By default, the CTRL/O function remains in effect until another CTRL/O. For example:

```
PUT SKIP LIST('Phase 1 complete... beginning phase 2...')
          OPTIONS (CANCEL_CONTROL_O);
```

If program output had been suspended by CTRL/O before this PUT statement executes, the PUT statement cancels the effect of the CTRL/O and outputs the data list.

7.2.2 FAST_DELETE Option

The FAST_DELETE option specifies, for a record in an indexed sequential file with alternate indexes, that only the current index for the file is to be updated.

The alternate indexes for the deleted record are not updated until the next time access is attempted to the record through the alternate index.

■ Rules

1. The FAST_DELETE option is valid only on a DELETE statement.
2. This option applies only to indexed sequential files.

■ Usage

This option can improve the speed of deletions when an indexed sequential file is updated.

7.2.3 FIXED_CONTROL_FROM Option

The FIXED_CONTROL_FROM option specifies a value to be written in the fixed control portion of a record in a file with variable-length records and a fixed control area. The format of the option is:

FIXED_CONTROL_FROM (variable-reference)

7.2.4 FIXED_CONTROL_TO Option

The FIXED_CONTROL_TO option specifies that the contents of the fixed control area of a record in a file with a fixed control area are to be assigned to a specified variable. The format of the option is:

FIXED_CONTROL_TO (variable-reference)

variable-reference
Specifies the variable associated with the fixed control area. The variable can be a scalar or a connected aggregate variable. It must not be an unaligned bit string or an aggregate consisting entirely of unaligned bit-string variables.

■ Rules

1. The FIXED_CONTROL_TO option is valid only on a READ statement.
2. The file must have variable-length records with a fixed-length control area and must be opened with the INPUT attribute and with the ENVIRONMENT option FIXED_CONTROL_SIZE_TO.
3. If the file is an existing file, the length of the variable must match the length of the fixed control area. If the length is not correct, the ERROR condition is signaled.

7.2.5 INDEX_NUMBER Option

The INDEX_NUMBER option specifies the particular index in an indexed sequential file to which a KEY option applies (primary index, secondary index, and so on).

The format of this option is:

INDEX_NUMBER (integer-expression)

integer-expression
Specifies the index to use. The value of expression must be the number of an index for records in an indexed sequential file. The primary index is zero, the secondary index is one, and so on.

■ Rules

1. The INDEX_NUMBER option is valid on a READ, REWRITE, or DELETE statement.
2. The file must be an indexed sequential file, and the KEY option must also be specified on the statement.

In the following example, STATE_FILE's third alternate key (that is, index number three) is a fixed binary population value:

```
DECLARE 1 STATE,
          2 NAME CHARACTER(20),          /* Primary Key */
          2 POPULATION FIXED BINARY(31),/* index #3*/
          2 CAPITAL,
          ,
          ,
        SIZE FIXED BINARY(31),
        STATE_FILE FILE RECORD INPUT KEYED SEQUENTIAL;
   ,
   ,
GET SKIP LIST(SIZE) OPTIONS(PROMPT(
       'Population value: '));
READ FILE(STATE_FILE) INTO(STATE) KEY(SIZE)
       OPTIONS(MATCH_GREATER,INDEX_NUMBER(3));
```

This READ statement obtains the record for the state whose population is greater than the value entered for the GET statement. For example, a value may be entered in response to this prompt as follows:

```
Population value: 8000000
```

In this case, the READ statement would read the first record in the index numbered three whose key value is greater than 8000000.

7.2.7 MATCH_GREATER_EQUAL Option

The MATCH_GREATER_EQUAL option specifies that the record of interest is the record whose key matches the key specified in the KEY option or, if no match is found, the first record whose key is greater than the key specified.

■ Rules

1. The MATCH_GREATER_EQUAL option is valid on the READ, REWRITE, and DELETE statements.
2. The KEY option must also be specified.
3. The file must be an indexed sequential file or a relative file.
4. MATCH_GREATER_EQUAL conflicts with the MATCH_GREATER option.

When MATCH_GREATER_EQUAL has been specified, it remains in effect for all subsequent keyed accesses of the file, until overridden by its specification with a false Boolean value or by the MATCH_GREATER option.

7.2.10 PROMPT Option

The PROMPT option specifies, when the input device is a terminal, a character-string prompt to be displayed prior to actual input. The format of this option is:

PROMPT (string-expression)

string-expression
Specifies a 1- to 254-character string expression.

■ Rules

1. The PROMPT option is valid only on a GET statement.
2. This option is meaningful only when the input device is a terminal.

■ Usage

Unlike a PUT statement followed by a GET statement, a GET statement with the PROMPT option is actually executed as a single statement. For example:

```
GET LIST (NUM) OPTIONS (PROMPT('Enter number: '));
```

When this statement is executed, the terminal display would be as follows:

```
Enter number: 44(RET)
```

The prompting string and the input data occur in the same statement.

On a terminal, use of the PROMPT option provides the following benefits:

1. If the display of the prompting string is interrupted, for example, by a broadcast message, the entire string is redisplayed following the message that interrupted it.
2. If CTRL/U or CTRL/R is entered in response to the prompt, the prompt message is repeated until data is entered.

The PROMPT option causes any data that was not processed by the last GET operation to be ignored. If the SKIP option is not specified, the prompt is output at the current cursor position. If the SKIP option is specified in conjunction with the PROMPT option, the SKIP operation is performed before the prompting message is displayed.

■ Usage

The following example illustrates a record whose record identification is saved for a later access of the file:

```
DECLARE BOOKFILE FILE RECORD KEYED,
        INBUF CHARACTER(180) VARYING,
        SAVE_RECORD_ID(2) FIXED BINARY(31),
        KEYVALUE CHARACTER(10);
    .
    .
OPEN FILE(BOOKFILE) ENV(RECORD_ID_ACCESS);
READ FILE(BOOKFILE) INTO(INBUF) KEY(KEYVALUE)
          OPTIONS(RECORD_ID_TO(SAVE_RECORD_ID));
    .
    .
CLOSE FILE(BOOKFILE);
    .
    .
OPEN FILE(BOOKFILE) INPUT ENV(RECORD_ID_ACCESS);
READ FILE(BOOKFILE) INTO(INBUF) OPTIONS(
    RECORD_ID_FROM(SAVE_RECORD_ID));
```

During the first open of the file, the record identification of a specified record is obtained and saved. When the file is subsequently reopened, this value is used to access a record and to effectively position the file at that record.

7.2.13 RECORD_ID_TO Option

The RECORD_ID_TO option specifies the name of a variable to be assigned the value of the record identification of the record on which the current operation is being performed.

The format of this option is:

RECORD_ID_TO (variable-reference)

variable-reference
Is a reference to a two-element array variable to receive the value of the record's identification.

The variable must be declared as (2) FIXED BINARY(31) and it must be connected.

■ Rules

1. The RECORD_ID_TO option is valid on the READ, WRITE, and REWRITE statements.
2. The file on which the operation is being performed must have been opened with the RECORD_ID_ACCESS option of the ENVIRONMENT attribute.

Chapter 8
File-Handling Built-In Subroutines

In addition to the PL/I input and output statements and the functions and features available through the options of the ENVIRONMENT attribute, there are also several built-in file-handling subroutines. These subroutines invoke VAX-11 RMS procedures. They are called built-in subroutines because you do not need to declare them before using them in a PL/I program. These subroutines are summarized in Table 8-1. They are described individually beginning in Section 8.1.

Table 8-1: Summary of File-Handling Built-In Subroutines

Subroutine	Function
DISPLAY	Returns information about a file.
EXTEND	Allocates additional disk blocks for a file.
FLUSH	Requests the file system to write all buffers onto disk to preserve the current status of a file.
NXTVOL	Begins processing the next volume in a multivolume tape set.
REWIND	Positions a file at its beginning or at a specific record.
SPACEBLOCK	Positions a file forward or backward a specified number of blocks.

8.1 DISPLAY Built-In Subroutine

The DISPLAY built-in subroutine returns information about a specified file. Its calling sequence is:

```
CALL DISPLAY (file-reference,variable-reference) ;
```

file-reference
Specifies the file variable or constant for which information is to be obtained. If the file is not currently open, the DISPLAY subroutine implicitly opens the file with the attributes specified in the declaration of the file.

variable-reference
Specifies the name of a structure variable into which information about the file is to be placed.

■ ENVIRONMENT Option Values Returned by DISPLAY

Table 8-2 summarizes the values returned by DISPLAY that correspond to ENVIRONMENT options and the data type of each structure member. For information on any of these ENVIRONMENT options, see the description of the option in Chapter 6, "ENVIRONMENT Options."

Table 8-2: ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
APPEND	BIT(1)	APPEND option is enabled/disabled
BATCH	BIT(1)	BATCH option is enabled/disabled
BLOCK_BOUNDARY_FORMAT	BIT(1)	Records can cross block boundaries
BLOCK_IO	BIT(1)	File is opened for block I/O
BLOCK_SIZE	FIXED BIN(31)	Block size of file (disk files only)
BUCKET_SIZE	FIXED BIN(31)	Bucket size of file (disk files only)
CARRIAGE_RETURN_FORMAT	BIT	Records have carriage return carriage control
CONTIGUOUS	BIT(1)	CONTIGUOUS option is enabled/disabled
CONTIGUOUS_BEST_TRY	BIT(1)	CONTIGUOUS_BEST_TRY option is enabled/disabled
CREATION_DATE	BIT(64)	Creation date of file
CURRENT_POSITION	BIT(1)	CURRENT_POSITION option is enabled/disabled
DEFERRED_WRITE	BIT(1)	DEFERRED_WRITE option is enabled/disabled
DELETE	BIT(1)	DELETE option is enabled/disabled
EXPIRATION_DATE	BIT(64)	Expiration date (magnetic tape files only)
EXTENSION_SIZE	FIXED BIN(31)	Current extension size (disk files only)
FILE_ID	(6)FIXED BIN(31)	File identification (disk files only)
FILE_SIZE	FIXED BIN(31)	File allocation (disk files only)
FIXED_CONTROL_SIZE	FIXED BIN(31)	Size of fixed control area
FIXED_LENGTH_RECORDS	BIT(1)	File has fixed-length records
GROUP_PROTECTION	CHAR(4) VARYING	Protection for group members
IGNORE_LINE_MARKS	BIT(1)	IGNORE_LINE_MARKS option is enabled/disabled
INDEX_NUMBER	FIXED BIN(31)	Current index number
INDEXED	BIT(1)	File is an indexed sequential file
INITIAL_FILL	BIT(1)	INITIAL_FILL option is enabled/disabled

(continued on next page)

■ File Attribute Information Returned by DISPLAY

Table 8–3 summarizes the file attribute information returned by DISPLAY, including:

- PL/I file description attributes and options specified for the file
- The file's organization, expanded file specification, and, if the file is an indexed sequential file, the number of keys it has

All names in Table 8–3 are second-level members of the structure PLI_FILE_DISPLAY.

Table 8–3: File Attribute Information Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
COLUMN_NUMBER	FIXED BIN(31)	Current column (stream output files only)
DIRECT	BIT(1)	File has/does not have DIRECT attribute
EXPANDED_TITLE	CHAR(128) VARYING	Expanded file specification
FILE_ORGANIZATION	CHAR(3)	SEQ, REL, or IDX
FORTRAN_FORMAT	BIT(1)	File has/does not have FTN (ASA) carriage control
INPUT	BIT(1)	File has/does not have INPUT attribute
KEYED	BIT(1)	File has/does not have KEYED attribute
LINE_NUMBER	FIXED BIN(31)	Current line number (stream output files only)
LINESIZE	FIXED BIN(31)	File's line size (stream output files only)
NUMBER_OF_KEYS	FIXED BIN(31)	Number of keys (indexed sequential files only)
OUTPUT	BIT(1)	File has/does not have OUTPUT attribute
PAGE_NUMBER	FIXED BIN(31)	Current page number (PRINT files only)
PAGESIZE	FIXED BIN(31)	Page size (PRINT files only)
PRINT	BIT(1)	File has/does not have PRINT attribute
RECORD	BIT(1)	File has/does not have RECORD attribute
SEQUENTIAL	BIT(1)	File has/does not have SEQUENTIAL attribute
STREAM	BIT(1)	File has/does not have STREAM attribute
UPDATE	BIT(1)	File has/does not have UPDATE attribute

Table 8-4 (Cont.): Device Information Returned by DISPLAY

Member Name	Meaning
SPL	Device is/is not spooled
SQD	Device is/is not sequential block-oriented (magnetic tape)
SWL	Device is/is not currently software write-locked
TRM	Device is/is not a terminal
WCK	Device performs write checking

8.2 EXTEND Built-In Subroutine

The EXTEND built-in subroutine increases the amount of space allocated to a disk file. Its calling sequence is:

CALL EXTEND (file-reference,integer-expression) ;

file-reference
Specifies the name of a file variable or constant associated with the file that is to be extended. If the file is not currently opened, the EXTEND subroutine opens the file with the OUTPUT attribute in order to extend it.

integer-expression
Is a fixed binary expression in the range of 0 to 4,294,967,295, specifying the number of 512-byte disk blocks to add to the file. If 0 is specified, PL/I uses the default extension quantity for the file.

To specify a value larger than 2,147,483,647 (the largest value that can be contained in a fixed binary integer in PL/I), you must express the number as a negative value; RMS inteprets the number as an unsigned integer.

■ Usage

Use the EXTEND built-in subroutine to explicitly extend a file during processing. Normally, RMS extends a file automatically, using a current extension size value, whenever an output operation causes a file to exceed its allocated space. The default value that RMS uses to extend a file is set by the ENVIRONMENT option EXTENSION_SIZE.

You can improve the performance of a program that is going to add a large number of records to a file by an explicit call to EXTEND before adding records to the file. If the call to EXTEND occurs before records are added, then RMS does not need to extend the file during the actual I/O operations.

8.5 REWIND Built-In Subroutine

The REWIND built-in subroutine positions a file so that the next record to be read will be the first record in the file or index. Its calling sequence is:

```
CALL REWIND (file-reference) ;
```

file-reference
Specifies the name of the file constant or file variable associated with the file to be rewound. If the file is not currently open, the REWIND subroutine implicitly opens the file with the attributes specified in the declaration of the file.

■ Usage

Use this subroutine to begin processing a file at its logical beginning. This subroutine is valid for disk files of all organizations and for sequential files on tape volumes. The position of the file following the call to the REWIND subroutine is as follows:

- If the file is a sequential file, the REWIND service positions the file to the first record in the file.
- If the file is a relative file, the REWIND service positions the file to the first occupied cell.
- If the file is an indexed sequential file, the REWIND service positions the file at the lowest key value in the current index.
- If the magnetic tape file is on a single volume, the volume is rewound. If the tape file exists on a multivolume tape set, the REWIND built-in subroutine rewinds the file to the beginning of the volume set.

You can also use the REWIND built-in subroutine to reposition a stream file after an end-of-file condition. Normally, if end-of-file (CTRL/Z on a terminal) is entered during an input operation on a stream input file, the PL/I program must close the input file and reopen it before it can read any more data. However, an ENDFILE ON-unit can be coded as follows:

```
ON ENDFILE(STREAMFIL) CALL REWIND(STREAMFIL);
```

This ON-unit calls the REWIND built-in subroutine each time an end-of-file is encountered for the file constant STREAMFIL. The REWIND built-in subroutine "repositions" the stream file at its beginning so that the program can continue reading input.

Chapter 9
File and Record Concepts

This chapter describes the following considerations for designing, creating, and using files:

- The file organization, that is, the physical arrangment of the records in the file
- The type of access that will be used to read, write, or update the records in the file, that is, whether the records will be accessed in sequential order or by a key
- The type of record in the file, that is, whether the records are variable length or fixed length
- The type of carriage control information, if any, used to print the records
- The format of stream files

Chapters 10 through 12 contain examples of creating and accessing files with different organizations and record formats in PL/I. For more detailed information on file design, see the *RMS-11 User's Guide*.

9.1 File Organizations

VAX-11 RMS supports three file organizations for record files. These are:

- Sequential
- Relative
- Indexed sequential

The relative and indexed sequential file organizations are valid only for disk devices. To read or write files on tape or unit record devices, you must use sequential organization.

The type of organization you select for a file and the attributes of the file, that is, the record format and size, the file size, and so on, are set when you create a file and need not be specified thereafter. When a program subsequently accesses an existing file, the file's organization and attributes are known to the file system.

9.2.1 Sequential Access

You can access records in a file with any file organization using sequential access. When you access a file sequentially, each read or write operation reads or writes the "next" record in the file.

As you process a file sequentially, PL/I always keeps track of the current record, that is, the record just read or written, and the next record, the record that follows the record just read or written.

When you access a relative file sequentially, the records are read or written in order by relative record number. In a file in which not all cells contain records, sequential input operations only involve cells that contain data records.

When you access an indexed sequential file sequentially, you may specify the number of the index on which to base the sequence. The "next" record in the input operation is the next ordered record in the specified index.

9.2.2 Random Access

When you access a file randomly by key, each input/output request must contain the KEY or KEYFROM option. The contents of the specified key depends on the file's organization, as follows:

- For a relative file, the key is the relative record number of the record to be accessed.
- For an indexed sequential file, the key is the portion of the record defined as a key field.
- In a disk file with fixed-length records, the key value is the relative record number of the record with respect to the beginning of the file. The first record in the file is relative record number 1, the second record is relative record number 2, and so on.

By default, a READ statement accesses a record based on an exact match of the key specified. In VAX-11 PL/I, you can optionally request that the READ statement match any record with an equal or greater key value, or any record with a greater key value.

9.2.3 Random and Sequential Access

When you access a file for random and sequential access, you can read records sequentially or randomly. For example, you can use a keyed READ statement to position the file at a specified record and then read or process records sequentially from that position.

9.3 Record Formats

VAX-11 RMS allows the following types of record format:

- Fixed-length records
- Variable-length records
- Variable with fixed-length control records

Fixed-length records and variable-length records are allowed for all file organizations. Variable with fixed-length control records are allowed in sequential and relative files only.

You need specify the format only when you create a file. Thereafter, each time you open the file PL/I determines the format of the records in the file.

9.3.1 Fixed-Length Records

In a file containing fixed-length records, all records have the same length. When you create a file with fixed-length records, you must specify the length of each record in the file; this size cannot be changed thereafter.

To create a file with fixed-length records in a PL/I program, use the FIXED_LENGTH_RECORDS option of the ENVIRONMENT attribute. The MAXIMUM_RECORD_SIZE option specifies the size of each record. For example:

```
DECLARE FIXED_FILE FILE RECORD KEYED OUTPUT
        ENVIRONMENT (
                     FIXED_LENGTH_RECORDS,
                     MAXIMUM_RECORD_SIZE(80));
```

When the file FIXED_FILE is opened, its record format is established as having fixed-length 80-character records.

When a file that has fixed-length records is processed by READ and WRITE statements, the file system checks the length of the variable specified in the INTO or FROM option to see if it is the same as the length of the records in the file. If not, the ERROR condition is signaled.

When you process a file with fixed-length records, you can specify the SCALARVARYING option of ENVIRONMENT to process records in the standard PL/I manner. For an example of using the SCALARVARYING option, see Section 6.2.43, "SCALARVARYING Option."

9.3.2 Variable-Length Records

In a file consisting of variable-length records, each record can have a different size. RMS places a count field at the beginning of each record to indicate its size; however, this count field is not considered a part of the data record, nor is the length of the count field included in the size of the record.

9.4 Carriage Control

VAX-11 PL/I provides a default carriage control for files that will be printed. This format, called carriage return format, may be specified in the ENVIRONMENT option list with the CARRIAGE_RETURN_FORMAT option; this option is never required.

When a file has carriage return format, the file can be output to a printer or terminal on a record-by-record basis. On output, each record is automatically preceded by a line feed (<LF>) character and followed by a carriage return (<CR>) character; these characters are not stored in the record. Thus, each record in the file occupies one line of output. This type of carriage control is valid for any file or record organization.

An alternate form of carriage control, called PRINTER_FORMAT, provides more explicit control of the output format and printing. Using printer format, you can specify such things as overprinting, skipping multiple lines, and so on. In a PL/I program, you will almost never need to use printer format; the PUT statement provides the same functions when it outputs data to a file with the PRINT attribute.

For details on using printer format, see Section 6.2.36, "PRINTER_FORMAT Option."

9.5 Physical Organization of Stream Files

In a PL/I program, the GET and PUT statements can access only files that have the STREAM attribute. A file has the STREAM attribute if:

- The file was declared explicitly with a DECLARE statement and the STREAM attribute. Or, the file was declared explicitly with a DECLARE statement and with neither the STREAM nor the RECORD attribute.
- The file was specified in and opened implicitly by a GET or PUT statement.

Files that are declared with the STREAM attribute have the following characteristics:

- Sequential organization of records.
- Variable-length records, with the maximum length of either 132 (default) or the length defined by the LINESIZE option.
- When the attributes STREAM, OUTPUT, and PRINT appear in the same declaration, a fixed control area that contains formatting information for the output file (see "Print File" and "PRINT Attribute" in the *VAX-11 PL/I Encyclopedic Reference*).

Stream files contain only ASCII data. The ASCII format used to represent program data in a stream output file differs depending on the attributes given to the file. For example, the representation of character strings differs depending on the presence or absence of the PRINT attribute in the file declaration.

Chapter 10
Sequential Files

This chapter shows examples of some typical sequential file input/output operations on sequential disk files and on sequential devices, including magnetic tapes.

10.1 Creating a Sequential File

Whenever a PL/I program opens a file with the SEQUENTIAL OUTPUT attributes, VAX-11 PL/I normally creates a new sequential file. By default, records are 510-byte variable-length records. Each WRITE statement adds a new record to the file.

10.1.1 Appending Records to an Existing File

In VAX-11 PL/I, you can open a file with the APPEND option of ENVIRONMENT to add new records at the end of an existing sequential file. This overrides the default action of PL/I, which is to create a new version of an existing file when the existing file is opened for output. For example:

```
OPEN FILE(BIRD_FILE) OUTPUT SEQUENTIAL
                TITLE('BIRDS.DAT') ENV(APPEND);
WRITE FILE(BIRD_FILE) FROM (NEWDATA);
```

This OPEN statement opens the file BIRD_FILE and positions it at its current end-of-file. The WRITE statement adds a new record at the end of the file.

10.1.2 Superseding an Existing File

The VAX-11 PL/I ENVIRONMENT option SUPERSEDE lets you create a new version of a file each time you write it, deleting an existing version. By default, each time a specific file is written, VAX-11 PL/I gives it a new version number and does not replace the existing version. For example:

```
OPEN FILE(CONTROL) OUTPUT RECORD TITLE('CONTROL.DAT;1')
       ENVIRONMENT(SUPERSEDE);
```

This OPEN statement opens the file CONTROL.DAT;1. If this file already exists, it is deleted.

10.2.2 Tape Positioning

When an existing magnetic tape file is opened, it is by default rewound, if necessary, and positioned at its beginning. This positioning can be overridden in the following ways:

- If the APPEND option of ENVIRONMENT is specified and if the file is opened with the OUTPUT attribute, the tape is wound and positioned at the end of the specified file. The next WRITE statement adds a new record at the end of the existing file.
- The CURRENT_POSITION option of ENVIRONMENT causes the tape to remain at its current position when the next file is opened. Thus, if the file is in the middle of the tape, it is not rewound when the next OPEN statement is specified for the tape.

By default, when a file is closed, the tape remains positioned following the last record that was read or written. The ENVIRONMENT option REWIND_ON_CLOSE can override this action and position the tape at its beginning.

While the file is open, the program can call the REWIND built-in subroutine to rewind the tape to its beginning.

For example:

```
DECLARE TAPEFILE FILE;

OPEN FILE (TAPEFILE) OUTPUT RECORD ENVIRONMENT(APPEND);
WRITE FILE (TAPEFILE) FROM (NEWREC);
    .
    .
CLOSE FILE(TAPEFILE) ENVIRONMENT (REWIND_ON_CLOSE);
OPEN FILE(TAPEFILE) INPUT RECORD;
```

In this example, the file TAPEFILE is opened for output with the APPEND option. WRITE statements add new records at the end of the tape file. Then, the CLOSE statement specifies that the tape is to be rewound, and the next OPEN statement opens the file for input. The first READ statement reads the first record in the file.

10.2.3 Blocking a Magnetic Tape File

On a magnetic tape, a block is a unit consisting of an integral number of records. Because of the control information needed to separate records on a tape, operations on a tape can be improved by blocking.

To create a blocked tape file, you must open it with the ENVIRONMENT option BLOCK_SIZE. This option specifies the size of the blocks. RMS automatically performs the blocking necessary. For example:

```
OPEN FILE(TAPEFILE) ENVIRONMENT(
                      BLOCK_SIZE (2048),
                      MAXIMUM_RECORD_SIZE (512),
                      FIXED_LENGTH_RECORDS);
WRITE FILE(TAPEFILE) FROM (BIG_RECORD);
```

Following this open, each WRITE statement writes a single record; the file system buffers the records until it accumulates four records and transfers them, blocked, to the tape volume.

- When a file that spans two or more volumes is being read and the tape reaches end-of-tape, the magnetic tape ACP sends a message to the system operator requesting the operator to mount the next tape in the volume set.

A PL/I program can request that the next volume in a volume set be mounted, for either an input or an output operation, by calling the NXTVOL built-in subroutine. The NXTVOL subroutine is described in Chapter 8, "File-Handling Built-In Subroutines."

The physical process of volume switching, whether the switching is performed automatically by RMS or as a result of a call to the NXTVOL built-in subroutine, is transparent to the PL/I program. As a user, you may wish to function as an operator to receive the volume switching requests and to mount the volumes yourself. For a description of the procedure for handling volume switching, see the *VAX/VMS Command Language User's Guide.*

10.3 Allocated and Spooled Devices

VAX/VMS spools low-speed input/output devices such as printers by accumulating data for the device in a file, and then queueing the file for processing when it is closed.

In a PL/I program, when you specify a device name such as LPA0: in a TITLE option, the specified device may be currently allocated for use by another user or be spooled. Depending on the status of the device, the following can occur:

- If the device is spooled, all output to the device is written to a temporary file. When the file is closed, it is submitted to the queue for the spooled device.
- If the device is allocated to another user, the UNDEFINEDFILE condition is signaled. If referenced in an ON-unit for this condition, the ONCODE built-in function returns the value associated with the status code SS$_DEVALLOC.
- If the device is allocated to the current process, PL/I assigns a channel to the device and each WRITE statement writes a physical line to the device.
- If the device is not allocated and is not spooled, PL/I assigns a channel to the device. This assignment performs an explicit allocation of the device to the current process.

You can allocate a device before running a program by issuing the DCL command ALLOCATE. Within a PL/I program, you can invoke the system service SYS$ALLOC to allocate a device. For information on commands for device allocation and control, see the *VAX/VMS Command Language User's Guide*. For information on allocating devices using the SYS$ALLOC system service, see the *VAX/VMS System Services Reference Manual.*

Chapter 11
Relative Files

This chapter describes considerations for creating and using relative files and shows examples of some typical relative file input/output operations.

11.1 The Organization of a Relative File

The relative file organization is suitable for files with data that can be arranged serially and uniquely identified by an integer value, for example, a part number or an employee identification number. Within the file, records are written into cells that are numbered. There is a one-to-one correspondence between the cell number and the integer value associated with the data in the record. This number is called the relative record number; the relative record number is the key by which records are written and accessed.

Figure 11–1 illustrates a relative file in which not all cells contain records. The first record written to the file was relative record number one (which may have been data for a part numbered one or an employee whose number is one, for example). The second record written was relative record number two. The third record written was relative record number four; thus cell number three does not contain a record.

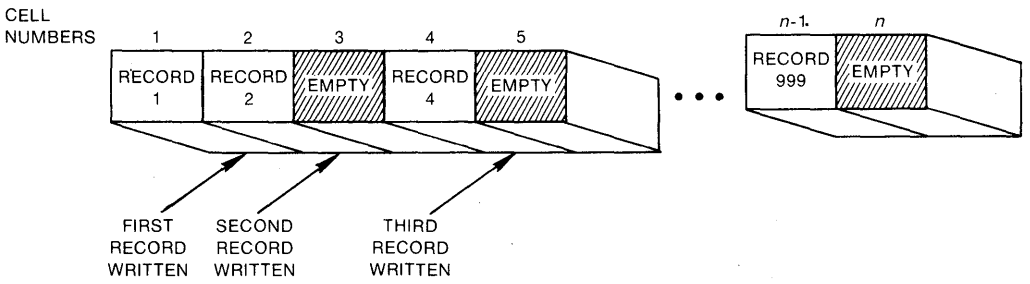


Figure 11–1: A Relative File

Although the cells in a relative file have the same length, the records need not be fixed-length records. However, when a record is smaller than the length of a cell, the unused space is wasted.

11.2.2 Record Size

When you specify the length of the records in a file, RMS uses the value you specify in the MAXIMUM_RECORD_SIZE option to calculate a cell size. It uses the following formulas to calculate the size:

Fixed-Length Records
cell-size = 1 + record-size

One byte is required for overhead; this byte contains a deletion indicator.

Variable-Length Records
cell-size = 3 + maximum-record-size

Three bytes are required for overhead; two bytes for the individual record size, and one byte for a deletion indicator.

When you select a record size for a relative file, you should try to specify a size that is no greater than the largest record that will be written. Otherwise, any unused space in each cell will be wasted. If you do not specify a maximum record size for either fixed- or variable-length records, VAX-11 PL/I uses the default length of 480 bytes.

11.2.3 Bucket Size

A bucket is the storage unit for data in the file. Records are arranged in buckets, which consist of an integral number of physically contiguous 512-byte disk blocks. Within the bucket, records can cross block boundaries; however, records cannot cross bucket boundaries.

When VAX-11 RMS transfers data from a file, it transfers data a bucket at a time; thus, a large bucket size reduces the number of actual data transfers that are required. When you do not specify a bucket size, RMS uses the cell size rounded to a multiple of 512 bytes. When records are written to the file, RMS places as many records as will fit in each bucket. Excess space is wasted.

You can improve I/O performance by specifying a bucket size that is a multiple of the cell size, and doing some simple calculations to determine whether space is being wasted. For example:

```
DECLARE EMP_FILE OUTPUT RECORD ENVIRONMENT (
                FIXED_LENGTH_RECORDS,
                MAXIMUM_RECORD_SIZE (80)
                BUCKET_SIZE (4) );
```

In this example, the file EMP_FILE will be created with 81-byte cells and buckets that are 2048 bytes (that is, four 512-byte blocks). Each bucket can contain 25 81-byte cells; 23 bytes in each bucket are unused.

When you specify a maximum record size and a bucket size for a relative file, you should consult the description of the BUCKET_SIZE option in Chapter 6, "ENVIRONMENT Options." That description contains formulas for calculating the bucket size within the limits required by RMS.

The following notes are keyed to Sample Program 11-1:

1. The structure PARTLIST describes the layout of the records in the file. The records will be ordered in the relative file according to part number, that is, using the field PARTLIST.NUMBER.
2. The file OLDFILE is the sequential file containing the records to be copied to a relative file. When the end-of-file is reached, the STOP statement terminates the program.
3. The relative file PARTS is declared with a maximum record number of 600. It has fixed-length, 38-byte records.
4. As each record is read into the structure PARTLIST, the value of NUMBER is copied to the fixed binary integer RECORD_NUMBER. The part number is maintained in each record in its character-string form.
5. Each WRITE statement copies the record to the output file, specifying the value of the part number as a relative record number.

Records in this file can subsequently be accessed either sequentially or by part number. To access a record by part number, you specify the number as a key. For example:

```
GET LIST(INPUT_NUMBER) OPTIONS(PROMPT('Part? '));
READ FILE(PARTS) INTO(PARTLIST) KEY(INPUT_NUMBER);
```

Here, the value entered in response to the GET statement is used as a key value to access a record in the file.

11.3.1 Populating a Relative File

In the example in the preceding section, the file PARTS is created by the opening of the file with the KEYED and OUTPUT attributes. When this program executes, the amount of space allocated for the file PARTS depends on the relative record numbers of the records that are written to the file. For example, if the largest record number specified for any record in the file is 600, but the largest record number specified for a record is 200, then RMS allocates only as much space as is needed for 200 records.

When you initially populate a file and you plan to fill the entire file, through the maximum record number, you can cause RMS to allocate space for the entire file using either of the following techniques:

- Specify the FILE_SIZE option to allocate space for the file when it is created, as described in Section 11.2.4, "File Size."
- Write the record with the largest relative record number first. This will force RMS to allocate space for the entire file.

These techniques can optimize the throughput for the subsequent file additions, since RMS will not need to perform repeated extensions to the file as records are added.

11.3.4 Error Handling

PL/I signals the KEY condition when errors occur while processing record numbers for relative files. For example, it signals the KEY condition when a relative record number exceeds the maximum record number specified for the file, or when the number of a record that already exists is specified in a KEYFROM option in a WRITE statement.

The sample ON-unit below shows how to detect whether a record already exists in a relative file or whether a record number specified exceeds the file's maximum record number.

```
ON KEY(PARTS) BEGIN;
DECLARE (RMS$_REX, RMS$_MRN) GLOBALREF FIXED BINARY(31) VALUE;
/* Check for duplicate records */
     IF ONCODE() = RMS$_REX THEN DO;     /* if duplicate */
        PUT SKIP EDIT('Part number',
                      PARTLIST.NUMBER,'exists. Reenter')
                (A,X,A,X,A);
        GET LIST(PARTLIST.NUMBER);       /* Get new value */
        GOTO GET_DATA;                   /* Go get other data */
        END;
/* Check for maximum record number exceeded */
     ELSE IF ONCODE = RMS$_MRN THEN DO;
        PUT SKIP EDIT('Part number',PARTLIST.NUMBER,
                      'invalid. Reenter')
                   (A,X,A,X,A);
        GET LIST(PARTLIST.NUMBER);       /* Get new value */
        GOTO GET_DATA;                   /* Go get other info */
        END;
   END;
   .
   .
GET_DATA:
```

In this example, the ON-unit declares symbolic names for two specific status values returned by ONCODE:

- The value RMS$_REX indicates that a record already exists.
- The value RMS$_MRN indicates that a relative record number specified exceeds the maximum record number.

In an ON-unit for the KEY condition for a relative file, ONCODE may also return the values associated with the following status codes:

- RMS$_RNF, which indicates that there is no record in the file with the relative record number specified by a KEY option.
- RMS$_KEY, which indicates that a key value is invalid, for example, if it is not an integer.

The symbolic names for these status codes must be declared with the GLOBALREF and VALUE attributes because the names are defined as global symbols by the VAX/VMS system. For more information on defining symbols and using symbols in ON-units, see Chapters 15, "Global Symbols," and 17, "Error and Condition Handling."

Chapter 12
Indexed Sequential Files

This chapter describes considerations for creating and using indexed files and shows examples of some typical operations on indexed sequential files.

12.1 Indexed File Organization

In an indexed sequential file, the file contains data records and pointers to the records. Data records and record pointers are arranged in buckets, which consist of an integral number of physically contiguous 512-byte disk blocks.

Individual records within the file are located by the specification of the keys associated with the records. Each file must have a primary key; this is a field within the record that has a unique value to distinguish it from all other records in the file. An indexed sequential file can also have up to 254 alternate keys, which need not have unique values.

As RMS writes records to an indexed file, it writes them in collating sequence according to the primary key, in buckets that are chained together. Thus, the file can be accessed sequentially using any key.

Figure 12–1 illustrates an indexed sequential file with a single key, or index.

The records in the file illustrated in Figure 12–1 consist of address data that might have been defined in a PL/I structure as follows:

```
DECLARE 1 ADDRESS_FILE,
          2 EMPLOYEE_NAME CHARACTER(30),
          2 ADDRESS,
            3 STREET CHARACTER(20),
            3 ZIP_CODE CHARACTER(5);
```

In this file, the key is the employee name.

When RMS writes records to an indexed sequential file, it builds and maintains a tree-like structure of key value and location pointers. When records are accessed by key, RMS uses the tree to locate individual records. Thus, when a PL/I program wants to access the record whose key value is JONES, RMS traverses the indexes to locate the record.

When new records are added to an indexed sequential file, a data bucket may not have enough room to accommodate a new record. In this case, RMS performs what is called bucket splitting — it inserts a new bucket in the chain of data buckets and moves enough records from the previous bucket to preserve the primary key sequence. Bucket splitting is transparent to the PL/I program; the program only knows that it has added a record to the file.

12.2 Creating an Indexed Sequential File

To create an indexed sequential file for VAX–11 PL/I, you must use the RMS–11 utility program DEFINE. After you create the file, you can use PL/I to populate the file by opening it with the UPDATE attribute and using WRITE statements to write records to it.

To invoke the DEFINE utility, enter the following command:

```
$ MCR DEF
```

This command invokes the RMS–11 utility by its task name, DEF. This utility is interactive: it prompts you to enter data and responds with error messages when you enter data incorrectly. It also provides information when you enter a question mark (?) in response to any of its prompts.

The short example below shows how to create the indexed sequential file that contains the records for the address file in Figure 12–1. The file will be named ADDRESS.DAT, and its character-string key field will be defined as the first 30 bytes of each record. Note that the only information that you must specify is:

- The file specification of the file you are creating
- IDX, to indicate that the file is an indexed sequential file
- The position of the key within the file's records
- The size of the key

12.3 Defining Keys

An indexed sequential file must have at least one key. It can have up to 255 keys; however, for file processing efficiency it is recommended that no more than seven or eight keys be defined. The time required to insert a new record or update an existing record is directly related to the number of keys defined. The retrieval time for an existing record is unaffected by the number of keys.

When you design an indexed sequential file, you must define each key in the following terms:

- The position and size of the key
- The data type of the key
- The index number of the key
- Key options selected for the key

In the example in the preceding section, only one key is defined, beginning in the first field of the record. However, when you want to define more than one key, or to define keys of different data types, you must be careful when you specify the key fields. The next few subsections describe some considerations for specifying keys.

12.3.1 Specifying Key Position and Size

When you specify a key, you must specify its position in the record and its length. The position must be specified with respect to the beginning of the record — thus, a key that is positioned beginning in the first byte of the record has a starting position of 0, a key positioned beginning in the 21st byte has a key position of 20, and so on.

To determine the key positions for fields within a structure, you can examine the storage map in the program listing that defines the structure. Figure 12–2 illustrates the relationship between the key field definitions and the storage map offsets.

The keys in Figure 12–2 can be specified as follows for DEFINE:

```
IT'S TIME TO DEFINE THE PRIMARY KEY
ENTER DATA TYPE(STR):RET
ENTER POSITION OF KEY:0RET ←
ENTER SIZE OF KEY:20RET ←
ENTER NAME OF KEY(NONE):RET
WILL YOU ALLOW DUPLICATE KEYS(NO)?RET

DO YOU WANT TO DEFINE MORE KEYS(NO)?YRET ←
ENTER DATA TYPE(STR):RET
ENTER POSITION OF KEY:116RET ←
ENTER SIZE OF KEY:30RET ←
ENTER NAME OF KEY(NONE):RET
WILL YOU ALLOW DUPLICATE KEYS(YES)?RET
WILL YOU ALLOW KEYS TO CHANGE(YES)?RET
DO YOU WISH TO DEFINE A NULL KEY VALUE(NO)?RET

JUST FINISHED ALTERNATE KEY NUMBER 1
DO YOU WANT TO DEFINE MORE KEYS(NO)?YRET ←
ENTER DATA TYPE(STR):RET
ENTER POSITION OF KEY:146RET ←
ENTER SIZE OF KEY:30RET ←
ENTER NAME OF KEY(NONE):RET
WILL YOU ALLOW DUPLICATE KEYS(YES)?RET
WILL YOU ALLOW KEYS TO CHANGE(YES)?RET
DO YOU WISH TO DEFINE A NULL KEY VALUE(NO)?RET

JUST FINISHED ALTERNATE KEY NUMBER 2
DO YOU WANT TO DEFINE MORE KEYS(NO)?YRET ←
ENTER DATA TYPE(STR):INTRET ←
ENTER POSITION OF KEY:20RET ←
ENTER SIZE OF KEY:4RET ←
ENTER NAME OF KEY(NONE):RET
WILL YOU ALLOW DUPLICATE KEYS(YES)?RET
WILL YOU ALLOW KEYS TO CHANGE(YES)?RET
DO YOU WISH TO DEFINE A NULL KEY VALUE(NO)?RET

JUST FINISHED ALTERNATE KEY NUMBER 3
DO YOU WANT TO DEFINE MORE KEYS(NO)?NRET ←
```

After all the keys are defined (that is, "N" is entered in response to the last question above), DEFINE begins prompting for file placement and allocation information, and then prompts for file protection information. You can press RET to answer all prompts. Or, you can study the file's requirements and specify placement and allocation information using the guidelines described in the *RMS–11 User's Guide*.

12.3.2 Key Data Types

Table 12–1 summarizes the valid data types for keys in VAX–11 RMS indexed sequential files, lists the corresponding PL/I data type declaration, and shows how to specify the key data type and length to the DEFINE utility.

12.3.4 Key Options

When you define alternate indexes for an indexed sequential file, you can specify:

- Whether duplicate keys are allowed. If you select the duplicate key option, multiple records in the file can have the same key value in the alternate index. If you do not allow duplicate keys, PL/I signals the KEY condition if you attempt to write a record with a duplicate key.
- Whether the key of a record can be changed. If you select the change option, a rewrite request can modify one or more key fields in the record. By default, PL/I signals the KEY condition if you attempt to rewrite a record in which a key field has been modified.
- Whether keys are to be initialized with null values. When a null value has been specified for a key and a record is inserted with the given key field equal to the null value, no index entry will be made in that alternate index.

These options are described in the *RMS-11 User's Guide*.

12.4 Using Indexed Sequential Files

After you have created an indexed sequential file with the DEFINE utility, you can write records to it by opening it with the UPDATE attribute and using PL/I WRITE statements. For example:

```
OPEN FILE(STATE_FILE) RECORD DIRECT UPDATE;
   .
   .
WRITE FILE(STATE_FILE) FROM(STATE) KEYFROM(STATE.NAME);
```

This WRITE statement writes the record whose key value is specified by the field STATE.NAME in the structure STATE.

When a WRITE statement adds a record to an indexed sequential file, the value of the KEYFROM option must always be the primary key. In fact, the WRITE statement causes the index number to be reset to zero if any other index number is in effect.

12.4.3 Accessing Records by Alternate Key

To read a record in an indexed sequential file using an alternate key, specify the INDEX_NUMBER option on a READ statement. For example, to access the record for a state whose flower is MAGNOLIA, the following statements could be written:

```
OPEN FILE(STATE_FILE) KEYED INPUT;
READ FILE(STATE_FILE) SET(STATE_PTR) KEY('MAGNOLIA')
      OPTIONS(INDEX_NUMBER(1));
```

The INDEX_NUMBER option specifies the first alternate index, the FLOWER field. The INDEX_NUMBER option is also valid on the REWRITE and DELETE statements.

You can access a file starting with an alternate index by opening the file with the INDEX_NUMBER option of ENVIRONMENT. For example:

```
OPEN FILE(STATE_FILE) SEQUENTIAL INPUT ENV(
              INDEX_NUMBER(2));
READ FILE(STATE_FILE) SET(STATE_PTR);
DO WHILE (^EOF);
    PUT SKIP EDIT(STATE,BIRD,'is the bird of',STATE,NAME)
                 (A,X,A,X,A);
    READ FILE(STATE_FILE) SET(STATE_PTR);
    END;
```

These statements, executed until the end-of-file is reached, access the records in the file STATE_FILE based on its second alternate index, the BIRD field.

12.4.4 Updating Records in an Indexed Sequential File

You can modify records in an indexed sequential file by opening the file with the UPDATE attribute and using REWRITE and DELETE statements to modify or delete records from the file.

The following example shows the correction of an invalid field in a record in the file STATE_FILE:

```
DECLARE (STATENAME,NEWNAME) CHARACTER(30) VARYING;
          .
          .
        OPEN FILE(STATE_FILE) KEYED SEQUENTIAL UPDATE;
        GET SKIP LIST(STATENAME) OPTIONS(PROMPT('State: '));
        READ FILE(STATE_FILE) SET(STATE_PTR) KEY(STATENAME);
        GET SKIP LIST(NEWNAME) OPTIONS(
                    PROMPT('New state flower name: '));
        STATE.FLOWER = NEWNAME;
        REWRITE FILE(STATE_FILE);
```

The REWRITE statement rewrites the current record in the file, that is, the record that was just read with the READ SET statement.

```
ON KEY(STATE_FILE) BEGIN;
DECLARE (RMS$_RNF, RMS$_DUP) GLOBALREF FIXED BINARY(31) VALUE;
/* Check for a record not found */
     IF ONCODE() = RMS$_RNF THEN DO; /* if record not found */
        PUT SKIP EDIT(STATENAME,'Not found.')
                 (A,X,A);
        STOP;
        END;
/* Check for duplicate key */
      ELSE IF ONCODE = RMS$_DUP THEN DO;
        PUT SKIP EDIT('Record already exists for',STATENAME)
                     (A,X,A);
        STOP;
        END;
END;
```

In this example, the ON-unit declares symbolic names for two specific status values returned by ONCODE:

- The value RMS$_RNF indicates that no record exists with the specified key value.
- The value RMS$_DUP indicates that a record already exists with the specified key in an index for which duplicate keys are not allowed.

In an ON-unit for the KEY condition, ONCODE may also return the value associated with the status code RMS$_KEY, which indicates that a key value is invalid, for example, if it is an incorrect data type.

The symbolic names for RMS status codes must be declared with the GLOBALREF and VALUE attributes because the names are defined as global symbols by the VAX/VMS system. For more information on defining symbols and using symbols in ON-units, see Chapters 15, "Global Symbols," and 17, "Error and Condition Handling."

Chapter 13
File Protection and File Sharing

This chapter provides examples of using ENVIRONMENT options to take advantage of special processing options of RMS. It includes discussions of:

- File protection
- File sharing

13.1 File Protection

Each user who is authorized to use the system is assigned a UIC (User Identification Code) by the system manager. When a PL/I program creates a file, the current UIC associated with the process executing the program defines the file's ownership.

Based on this UIC, called the owner UIC, the file system defines the protection of the file in terms of (1) which other users on the system can access the file and (2) what operations they can perform on the file. The other users in the system are defined as follows:

- Owner. Any other process that has the same UIC as that established as the file's owner is also the owner of a file.
- Group. A process that has the same group number in its UIC is a member of the owner's group.
- System. A process that has a group number in the system-defined range or that has the SYSPRV user privilege is in the system user category.
- World. All jobs and processes that do not fall into the other three categories belong to the world category.

13.1.2 Defining a File's Protection

When you specify ENVIRONMENT options for a file you are creating in a PL/I program, you can specify the following options to define the access permitted to various users:

OWNER_PROTECTION
GROUP_PROTECTION
SYSTEM_PROTECTION
WORLD_PROTECTION

These options specify the types of access permitted by the specification of the following codes

- R — gives the right to read the file.
- W — gives the right to modify the file.
- E — gives, for files containing executable program images, the right to execute the program.
- D — gives the right to delete the file.

These codes can be specified in any order for an option; if you specify an option and omit a code, that category of user is denied that type of access. If you specify one or more protection options, the protection for categories you do not specify defaults to no access. If you do not specify any protection options, then PL/I uses the current default protection for all the categories.

For example:

```
ENVIRONMENT (
             OWNER_PROTECTION ('RWE')
             SYSTEM_PROTECTION ('R')
             GROUP_PROTECTION('R'))
```

This specification defines protection to a file as follows:

- The OWNER_PROTECTION option specifies RWE, that is, read, write, and execute access. Because D is not specified, the owner is not allowed delete access and thus cannot inadvertently delete the file.
- The SYSTEM_PROTECTION and GROUP_PROTECTION options specify only read access for system and group users.
- The WORLD_PROTECTION option is not specified; this denies all access to all users who are in the world category.

Note that the DCL command SET PROTECTION allows the owner of a file to change the file's protection at any time. Additional commands and user privileges allow the protection of a file to be overridden or changed. For details on these commands and privileges, see the *VAX/VMS Command Language User's Guide*.

The file system applies the protection you specify for a file when the file is accessed from a program or from the DCL command level. It also applies the protection when the file is to be shared, as described in the next section.

of the file. Other processes may access the file only for reading; they must specify SHARED_WRITE, to indicate that they allow writing of the file while they are reading it.

If SHARED_WRITE is specified, processes that subsequently access the file with the SHARED_WRITE option may write the file.

Both the SHARED_READ and SHARED_WRITE options may be specified for a file.

Table 13–1 summarizes the effects of opening a file with file-sharing options.

Table 13–1: Effects of File-Sharing Options

Open Option and Access Specified by First Opener	Open Option Specified by a Subsequent Opener	Access Allowed Subsequent Opener
ENV(NO_SHARE)[1] INPUT, OUTPUT, or UPDATE	ENV(NO_SHARE) ENV(SHARED_READ) ENV(SHARED_WRITE)	None. The UNDEFINED-FILE condition is signaled[2]
ENV(SHARED_READ) INPUT	ENV(NO_SHARE)	None. The UNDEFINED-FILE condition is signaled[2]
	ENV(SHARED_READ)	The file is accessed for input
	ENV(SHARED_WRITE)	The UNDEFINEDFILE condition is signaled[2]
ENV(SHARED_READ) OUTPUT or UPDATE	ENV(NO_SHARE)	None. The UNDEFINED-FILE condition is signaled[2]
	ENV(SHARED_READ)	None. The UNDEFINED-FILE condition is signaled[2]
	ENV(SHARED_WRITE)	The file can be accessed for input only
ENV(SHARED_WRITE) INPUT	ENV(NO_SHARE)	The UNDEFINEDFILE condition is signaled[2]
	ENV(SHARED_READ)	The file can be accessed for input, output, or update
	ENV(SHARED_WRITE)	The file can be accessed for input, output, or update
ENV(SHARED_WRITE) OUTPUT or UPDATE	ENV(NO_SHARE)	None. The UNDEFINED-FILE condition is signaled[2]
	ENV(SHARED_READ)	None. The UNDEFINED-FILE condition is signaled[2]
	ENV(SHARED_WRITE)	The file can be accessed for input, output, or update

1. You must have write access privilege to open the file with the NO_SHARE option.
2. ONCODE returns the value for RMS$_FLK. See Section 13.2.2, "File Locking."

A record is locked when both of the following are true:

- A READ statement is issued for the record.
- The file containing the record was opened with the OUTPUT or UPDATE attribute.

A record remains locked until one of the following occurs:

- The locked record is rewritten or deleted.
- A READ, WRITE, REWRITE, or DELETE statement is executed to access another record in the same file.
- The REWIND built-in subroutine is called to rewind the file to its beginning.
- The file is closed.

Records are also locked for the duration of a WRITE, REWRITE, or DELETE statement to ensure that the I/O completes. The records are unlocked when these statements complete.

If a procedure in another process attempts to access a record that is locked, the ERROR condition is signaled. In an ON-unit that executes following this condition, a reference to the ONCODE built-in function returns the value associated with the RMS status code RMS$_RLK (meaning that the record is locked).

Thus, a file-sharing application can test whether a record in a file is currently locked in an ON-unit, as in the following example:

```
ON ERROR BEGIN;
DECLARE RMS$_RLK GLOBALREF FIXED BINARY(31) VALUE;

    IF ONCODE() = RMS$_RLK THEN CALL RECORDSYNC;
       ELSE CALL RESIGNAL();
END;
```

The ON-unit in this example tests whether any ERROR condition is signaled as a result of attempting to access a locked record. If so, it calls a procedure that will synchronize with the other process reading the record. Otherwise, it calls the RESIGNAL built-in subroutine to perform default condition handling.

Part III
Procedure Calling and Condition Handling

Chapter 14
Argument Passing

The architecture of the VAX–11 defines a set of conventions for passing arguments among procedures. These conventions make it possible for procedures that are written in PL/I to invoke, with a CALL statement or with a function reference, procedures written in other programming languages, for example FORTRAN, PASCAL, and assembly language. These conventions are known as the VAX–11 Calling Standard.

This chapter describes the calling standard and the argument-passing mechanisms defined in it, and explains the VAX–11 extensions to the PL/I language that support it.

This chapter assumes a knowledge of the PL/I conventions and rules for passing arguments to external procedures, as described in the *VAX–11 PL/I Encyclopedic Reference* under the heading "Parameters and Arguments." Although the argument-passing structures used by the system are transparent to your PL/I programs, they are presented in this chapter (in a simplified format) to provide you with the necessary background to write calls to non-PL/I procedures.

14.1 The Call Stack

A call stack is a temporary area of storage that the system allocates for each user process. On the call stack, the hardware maintains information about each block activation in the current image.

14.1.1 Call Frames

Each time a procedure block is activated in a PL/I program, the hardware creates a structure on the call stack. This structure is the call frame for the procedure invocation. The call frame for each active procedure contains:

- A pointer to the call frame of the previous block activation. This pointer is called the Frame Pointer (FP).
- The saved Argument Pointer (AP) of the previous procedure invocation.
- The address in storage of the point of invocation of the procedure, that is, the address of the next instruction following the CALL instruction or CALL

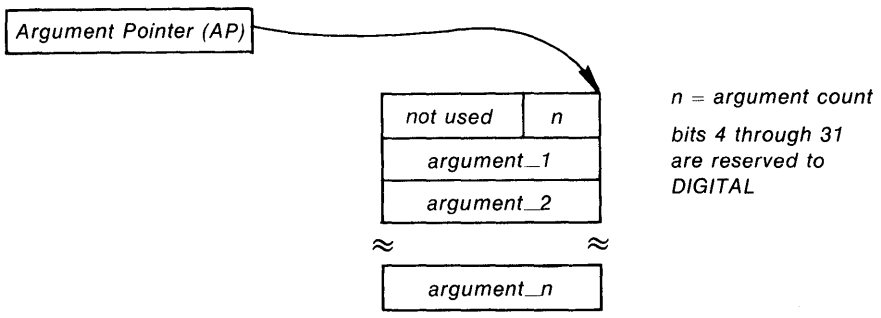


Figure 14–2: An Argument List

The calling standard defines three ways that data can be passed in an argument list. When you are coding a reference to a non-PL/I procedure, you must note the mechanism by which each argument is to be passed and write the parameter descriptor for each argument accordingly.

The three argument-passing mechanisms are:

- By immediate value. When an argument is passed by immediate value, the actual value of the argument is present in the argument list.
- By reference. When an argument is passed by reference, the address in storage of the argument is present in the argument list.
- By descriptor. When an argument is passed by descriptor, the address in storage of a data structure describing the argument is present in the argument list.

Sections 14.2 through 14.4 describe these argument-passing mechanisms in detail. These sections describe the arguments in terms of PL/I data types, dummy arguments created, if any, parameter-passing conventions, and attributes to define the manner in which parameters are to be passed. Figures 14–3 through 14–6, which accompany these sections, illustrate these mechanisms.

Remember that when PL/I creates a dummy argument, modifications, if any, that the called procedure makes to the dummy argument are not accessible to the caller.

Note that most of the examples show calls to VAX/VMS system service procedures. These examples do not describe the procedures themselves. For general and specific descriptions of system services, see the *VAX/VMS System Services Reference Manual.* For additional details on calling system services from PL/I programs, see Chapter 19, "System Services."

14.2 Passing Arguments by Immediate Value

You must use the VALUE attribute in a parameter descriptor for an argument to be passed by immediate value. The following declaration of the external

parameter descriptor is specified with the ANY and VALUE attributes, the data types of the dummy arguments that PL/I creates are:

Data Type of Written Argument	Data Type of Dummy Argument
FIXED BINARY, or FIXED DECIMAL (p,0)	FIXED BINARY (31)
BIT or BIT ALIGNED	BIT (32) ALIGNED
ENTRY	ENTRY
OFFSET	OFFSET
POINTER	POINTER

If a parameter descriptor is specified as VALUE with a particular data type (as opposed to being specified as ANY), a dummy argument of that data type is always created, and the written argument is assigned to the dummy. The written argument must be valid for conversion to the data type specified in the corresponding parameter descriptor.

14.3 Passing Arguments by Reference

By reference is the default argument-passing mechanism used by PL/I for all arguments except character strings and arrays with nonconstant extents. The parameter descriptor for an argument to be passed by reference need specify only the data type of the parameter.

For example, the Read Event Flags (SYS$READEF) system service requires its first argument to be passed by immediate value and its second argument to be passed by reference. This procedure can be declared as follows:

```
DECLARE SYS$READEF ENTRY (FIXED BINARY(31) VALUE,
                          BIT (32) ALIGNED);
```

When this procedure is invoked, the second argument must be a variable declared as BIT(32) ALIGNED. PL/I passes the argument by reference. Figure 14–4 illustrates argument passing by reference.

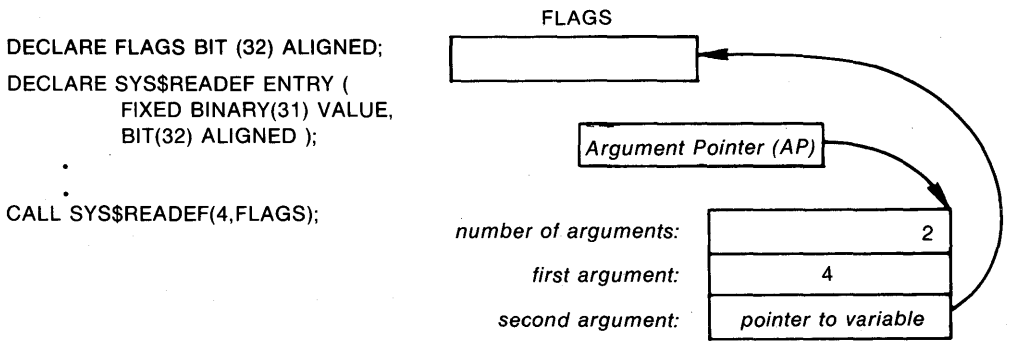


Figure 14–4: Argument Passing by Reference

a PL/I parameter descriptor with asterisk extents. In FORTRAN, arrays must always be passed by reference; the array's extents are, by custom, passed as separate arguments. The ANY attribute provides a convenient way to express an array parameter for FORTRAN, as in the following example:

```
FTNARRAY: PROCEDURE(X);
DECLARE SUM ENTRY (ANY, FIXED BINARY(31))
                    RETURNS (FLOAT);

DECLARE (S, X(*)) FLOAT;

S = SUM(X, DIM(X,1));
```

In this example, SUM is a FORTRAN procedure that sums the elements of a one-dimensional array of floating-point numbers. Its second parameter is the number of elements in the array.

14.3.3.2 Dummy Arguments for Arguments Passed by ANY — When a parameter that is declared with the ANY attribute but without the VALUE attribute is associated with a written argument that is a variable, PL/I places the address of the actual variable in the argument list. If the procedure is invoked with a constant or expression for this argument, PL/I creates a dummy argument and places the address of the dummy argument in the argument list.

In creating a dummy argument, PL/I performs the conversions listed below:

Data Type of Written Argument	Data Type of Dummy Argument
BIT (unaligned)	BIT ALIGNED
FIXED BINARY, or FIXED DECIMAL (p,0)	FIXED BINARY (31)
CHARACTER VARYING	CHARACTER nonvarying

In all other cases, the data type of the dummy argument is the same as the data type of the written argument.

14.3.4 Using Pointer Values for Arguments Passed by Reference

When an argument is passed by reference, PL/I places the address of the actual argument in the argument list. This address can be interpreted as a pointer value. In fact, you can explicitly specify a pointer value as an argument for data to be passed by reference. For example:

```
DECLARE SYS$READEF (ANY VALUE, POINTER VALUE),
        FLAGS BIT(32) ALIGNED;
CALL SYS$READEF (4, ADDR(FLAGS));
```

At this procedure invocation, PL/I places the pointer value returned by the ADDR built-in function directly in the argument list. Figure 14–5 illustrates the argument list for this example. Note that the actual argument list in this example corresponds to the argument list shown in Figure 14–4.

```
DECLARE UNSTRING ENTRY (CHARACTER(*)),
        TESTBITS ENTRY (BIT(3)),
        MODEST ENTRY (1,
                      2 CHARACTER(*),
                      2,
                        3 BIT(3),
                        3 BIT(3));
```

Figure 14–6 illustrates a character-string descriptor and shows how a character-string argument is passed by descriptor. This example illustrates the type of character-string descriptor used by system services; this descriptor does not contain additional information required by other classes of descriptor.

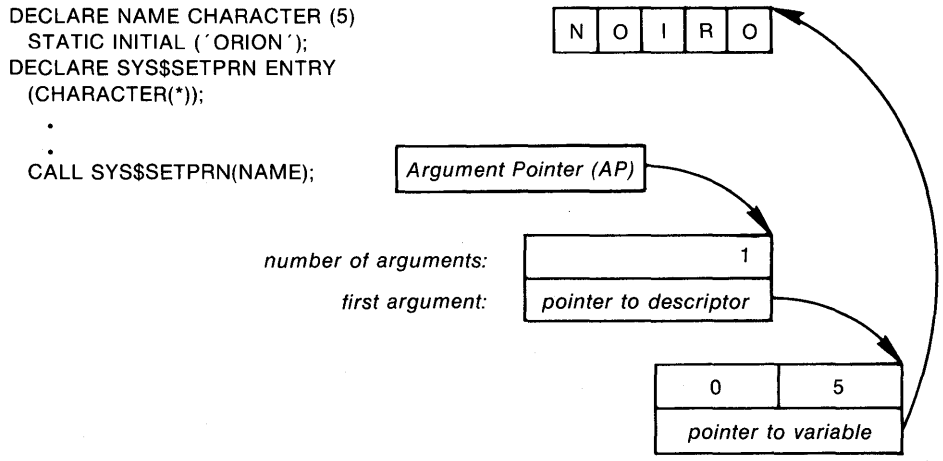


Figure 14–6: Argument Passing by Descriptor

14.4.2 Passing Character Strings

When you declare a non-PL/I procedure that requires a character-string descriptor for an argument, specify the parameter descriptor as CHARACTER(*). For example, the Set Process Name (SYS$SETPRN) system service requires the address of a character-string descriptor as an argument. You can declare this service as follows:

```
DECLARE SYS$SETPRN ENTRY (CHARACTER(*));
```

When a parameter is declared as CHARACTER(*), its written argument can be:

- A character-string constant or expression
- A fixed-length character-string variable
- A varying character-string variable or a variable declared as CHARACTER(*) VARYING

For any of these arguments, PL/I constructs a character-string descriptor and places its address in the procedure's argument list.

2. Declare a structure variable in your program whose members and attributes correspond to the structure declared in the parameter descriptor for the argument.
3. Assign values to the members of the structure variable providing the required information. For a character-string descriptor, you must provide the length of the string and a pointer to the variable containing its value.
4. Pass the name of the structure variable as an argument in the procedure invocation.

The Set Process Name (SYS$SETPRN) system service shown in Figure 14–7 requires a text name string to be passed by descriptor. The structure variable NAME_DESC is a character-string descriptor: its members describe the length and location of the character-string variable NEWNAME. The value of NEWNAME is the actual argument passed to the procedure. Note that the call in this example is equivalent to the example shown in Figure 14–6 of passing an argument by descriptor.

```
DECLARE SYS$SETPRN ENTRY (1,
                          2 FIXED BINARY(31), ❶
                          2 POINTER);

DECLARE 1 NAME_DESC,
          2 NAME_LENGTH FIXED BINARY (31), ❷
          2 NAME_ADDRESS POINTER;

DECLARE NEWNAME CHARACTER (5) STATIC INITIAL ('ORION');
NAME_DESC.NAME_LENGTH = LENGTH(NEWNAME); ❸
NAME_DESC.NAME_ADDRESS = ADDR(NEWNAME);

CALL SYS$SETPRN(NAME_DESC); ❹
```

Figure 14–7: Coding a Character-String Descriptor

Note that this example can be simplified by declaring SYS$SETPRN as follows:

```
DECLARE SYS$SETPRN ENTRY (ANY);
```

All other variables, and the procedure call, would be the same as in Figure 14–7.

14.4.3 Using the DESCRIPTOR Built-In Function

If a parameter descriptor specifies ANY without VALUE, a corresponding argument may be a reference to the DESCRIPTOR built-in function. For example:

```
DECLARE P ENTRY (ANY);
DECLARE (X,Y) FIXED DECIMAL (7,2);

CALL P(DESCRIPTOR(X));
CALL P(Y);
```

Here, X is passed by descriptor because the DESCRIPTOR built-in function so specifies. Y is passed by reference.

OPTIONS (VARIABLE). At least one parameter descriptor must be specified; the last parameter descriptor given in the ENTRY attribute is used for any extra arguments.

The Formatted ASCII Output (SYS$FAO) system service is an example of a procedure that has a variable-length argument list. It can be declared as follows:

```
DECLARE SYS$FAO ENTRY (CHAR(*), FIXED BINARY(15),
                       CHAR (*), ANY VALUE) OPTIONS (VARIABLE);
```

This parameter descriptor specifies only four arguments. When SYS$FAO is invoked with more than four arguments, PL/I uses the parameter descriptor of the last parameter (ANY VALUE) to pass all the additional arguments. If any argument that will be specified is not to be passed by value, you must specify a parameter descriptor for the argument in the declaration.

14.5.2 Optional Arguments

In the PL/I language, there can be no optional parameters to a PL/I procedure. You must always specify a written argument for each parameter in the entry declaration.

Many non-PL/I procedures with fixed-length argument lists accept optional arguments and provide a default action if no value or a value of zero is specified for the optional argument. When an optional argument is not specified, its corresponding argument list longword must contain a zero.

In PL/I, you can omit the specification of an optional argument in a written argument list as long as you enter the correct number of commas to ensure that the argument list will have the correct number of longwords. You can indicate that you are not specifying an optional argument in either of the following ways:

- Omit the argument from the argument list.
- If the argument is to be passed by immediate value, specify a zero for the written argument.

For example, an argument list that has three optional arguments can be written as follows:

```
(,,)
```

If the parameter descriptor of each argument specifies ANY VALUE, the argument list may also be written:

```
(0,0,0)
```

In either case, the called procedure must detect and interpret zeros in the argument list. The following example illustrates optional arguments omitted from an argument list:

```
DECLARE SYS$ASCTIM ENTRY (
                          ANY,CHAR(*),ANY) OPTIONS (VARIABLE),
        TIME_STRING CHARACTER(24);

    .
    .
CALL SYS$ASCTIM(,TIME_STRING,,);
```

Chapter 15
Global Symbols

In standard PL/I, a variable that is to be shared by external procedures must be declared with the EXTERNAL attribute in each procedure that references it. VAX-11 PL/I provides an alternate method for defining external variables. Using the GLOBALDEF attribute, one module may completely declare an external variable; all other modules that reference the variable declare it with the GLOBALREF attribute. The VALUE and READONLY attributes provide additional control over the storage of these variables.

Even if a PL/I program does not itself define external variables in this way, the GLOBALREF attribute permits a PL/I program to access variables defined in modules written in other languages.

This chapter describes:

- Using global symbols within PL/I procedures
- The READONLY and VALUE attributes
- Declaring and using system-defined global symbols

15.1 Using Global Symbols in PL/I Procedures

Within your PL/I programs, you can define variables as global external symbols when you are coding calls to system procedures. You can also use global symbols instead of external variables in PL/I procedures and functions.

Table 15–1 summarizes the differences between global symbols and external variables. Note that a primary difference between these variables is the manner in which the linker allocates storage for them. Linker storage allocation is described in Chapter 18, "Storage Allocation and Usage."

- Only one procedure in a program may declare a particular external variable with the GLOBALDEF attribute.

The GLOBALREF attribute indicates that the declared name is a global symbol defined in an external procedure.

The GLOBALREF attribute implies the EXTERNAL and STATIC attributes. The corresponding name must be declared in another procedure with the GLOBALDEF attribute or, if the external procedure is written in another programming language, its equivalent in that language.

The following restrictions apply to the use of the GLOBALREF attribute:

- The GLOBALREF attribute conflicts with the INITIAL, GLOBALDEF, and INTERNAL attributes.
- If GLOBALREF is specified with the FILE attribute, no other file description attributes can be specified.

15.1.2 Defining Global Symbols in PL/I

To create a global symbol definition in a PL/I program, you must declare it with the GLOBALDEF attribute in one, and only one, PL/I external procedure. The GLOBALDEF attribute implies the EXTERNAL attribute.

An external variable defined with the GLOBALDEF attribute can be accessed by external procedures that declare the name with the GLOBALREF attribute. For example, the procedure ABC contains:

```
ABC:  PROCEDURE;
   DECLARE UNIQUE_VALUE GLOBALDEF FIXED BINARY
       INITIAL (60);
   DECLARE XYZ EXTERNAL ENTRY (CHARACTER (*));
       .
       .
   CALL XYZ ('STRING');
```

The procedure XYZ contains:

```
XYZ:  PROCEDURE (STRING_VAL);
   DECLARE UNIQUE_VALUE GLOBALREF FIXED BINARY;
       .
       .
```

In the preceding example, the external variable UNIQUE_VALUE is declared with the GLOBALDEF attribute and initialized in the procedure ABC. The called external procedure XYZ declares this variable with the attribute GLOBALREF and the appropriate data type attributes.

15.1.3 Using MACRO Global Symbols with Multiple Definitions

Using the VAX-11 MACRO programming language, it is possible to give a global external variable more than one name. However, in a PL/I procedure, only one global symbol name may be used for a particular variable. PL/I assumes that distinct global symbol names denote distinct storage locations; the storage associated with different names may not overlap. This rule applies only to global symbols that are declared without the VALUE attribute.

4. All declarations of the variable must specify the VALUE attribute.
5. The variable is not addressable; thus it cannot be used as the argument of the ADDR built-in function.

A variable declared with the VALUE attribute can be specified as a value to initialize another variable; it must have the same data type as the variable that is being initialized. For example:

```
DECLARE TEMP GLOBALDEF FIXED VALUE INITIAL(10),
        ABC FIXED STATIC INIT(TEMP);
```

The declaration of ABC in this example gives ABC the value 10.

15.3 Obtaining Definitions for System Global Symbols

Within the VAX/VMS system, many global symbol definitions are used and accessed by programs and procedures in many ways. The most common uses are to define symbolic names for:

- Return status values from system procedures
- Function codes for system programs
- Symbolic names for system mailbox message senders
- Bit field definitions in system data structures

From a PL/I program, you can declare the symbolic names for system global symbols with the GLOBALREF and VALUE attributes. The format of these declarations is:

DECLARE symbol-name GLOBALREF FIXED BINARY(31) VALUE;

The GLOBALREF attribute indicates to PL/I that the variable is a reference to a global symbol defined in another module. The VALUE attribute indicates that the value of the variable is to be treated as if it were a constant.

The definitions for system global symbols are declared in the default system object module libraries. These libraries are automatically searched when you link a PL/I program. Of particular interest are the global symbols that define symbolic names for system service and file system return status values. Their use is described in the next chapter, "Return Status Values."

Chapter 16
Return Status Values

The VAX–11 mechanism for returning a status value among procedures is to return a fixed-point binary value in the general register R0. This value, called a return status value, indicates the success or failure of the operation performed by the called procedure.

In PL/I, passing a return status value in R0 is equivalent to a function return of a value declared as FIXED BINARY(31). In fact, when a PL/I function that returns a value that can be expressed in 64 bits or less executes a RETURN statement, PL/I places the return value specified in R0, and R1 if necessary, before returning control to the caller.

Thus, to obtain a return status value from any system procedure, you can declare the procedure as a function as shown in the following example:

```
DECLARE SYS$SETEF ENTRY (FIXED BINARY(31) VALUE)
                  RETURNS (FIXED BINARY (31));
```

This declaration of the SYS$SETEF procedure allows you to invoke the procedure as a function and to obtain a return status value.

This chapter provides information on:

- The format of a return status value, that is, the meaning of particular bits within the value
- Recommended techniques for testing a return status value for success or failure or for a specific condition

The information in this chapter also applies to the return status values signaled by PL/I run-time procedures. Error signaling and condition handling are described in detail in Chapter 17, "Error Signaling and Condition Handling."

16.1 Format of Return Status Values

All VAX/VMS system procedures and programs use a longword value to communicate specific return information. When a main procedure executing under the control of the DCL command interpreter executes a RETURN statement to return control to the command level, the command interpreter uses the return status value to display a message on the current output device.

module $STSDEF. This module is in the default PL/I text library PLISYSDEF.TLB (described in Section 2.4.7, "Default System INCLUDE Library"). The module $STSDEF contains the following declarations:

```
declare sts$value fixed binary(31),     /* status value */
        1 sts$fields based (addr(sts$value)),
          2 sts$severity,               /* low-order 3 bits */
            3 sts$success bit(1),       /* low-order bit */
            3 sts$rest bit(2),          /* bits 1 through 2 */
          2 sts$msg,                    /* bits 3 through 15 */
            3 sts$msg_no bit(12),       /* numeric value */
            3 sts$fac_sp  bit(1),       /* if set, facility specific */
          2 sts$fac,                    /* bits 16 - 27 */
            3 sts$fac_no bit(11),       /* facility number */
            3 sts$cust_def bit(1),      /* 0 = DIGITAL */
          2 sts$control,
            3 sts$inhib_msg bit(1),     /* 1 = do not print */
            3 sts$reserved bit(3),      /* 32 bits */
          2 sts$filler character(0);    /* for byte alignment */
```

To obtain these declarations, specify a %INCLUDE statement in a PL/I program as follows:

```
%INCLUDE $STSDEF;
```

The compiler will locate this module in PLISYSDEF automatically.

The next three sections describe the following ways you can use these variables:

- To test for successful or nonsuccessful completion of a procedure
- To test whether a procedure returned a specific value
- To determine, set, or display any field within a longword status value

Remember that you can test return status values from system procedures only if you declare the procedures as functions.

16.2 Testing for Success or Failure

To test a return status value for success or failure, you need only test the variable STS$SUCCESS declared in the structure STS$FIELDS. If this bit is true, it indicates that the return value is a successful value. For example:

```
DECLARE SYS$SETPRN ENTRY (CHARACTER(*))
                RETURNS (FIXED BINARY(31));
%INCLUDE $STSDEF;

STS$VALUE = SYS$SETPRN('Student');
IF ^STS$SUCCESS THEN GOTO BAD_NAME;
```

The statements at the label BAD_NAME can test the value of the variable STS$VALUE and take some action based on its value.

16.3 Testing for Specific Return Status Values

Each numeric return status value defined by the system has a symbolic name associated with it. The names of these values are defined as system global symbols, and their values can be accessed by referring to their symbolic names.

The next example illustrates the invocation of the Set Event Flag (SYS$SETEF) system service, followed by tests for (1) success or failure and (2) the successful status code SS$_WASSET.

```
DECLARE SS$_WASSET FIXED BINARY GLOBALREF VALUE,
        SYS$SETEF ENTRY (FIXED BINARY(31) VALUE);
%INCLUDE $STSDEF;

STS$VALUE = SYS$SETEF (4);

IF ^STS$SUCCESS THEN RETURN (STS$VALUE); ❶
IF STS$VALUE = SS$_WASSET THEN DO; ❷
      .
      .
```

In this example, the symbolic name SS$_WASSET is declared as a global symbol. The value associated with this return status is a successful value; it indicates that the flag specified in the procedure invocation was previously set.

The procedure invocation returns the status value in the variable STS$VALUE. The IF statement checks the variable STS$SUCCESS for success or failure. If the service returned a failure condition, the procedure returns with the value of STS$VALUE in the RETURN statement. If the service returned a successful status, the procedure continues with an IF statement that checks whether the flag was previously set. If so, the DO statement specified in the THEN clause activates the DO-group.

Note the effect of the RETURN statement in this example. If this procedure is the main procedure, the RETURN statement that specifies the current value of the variable STS$VALUE will cause the command interpreter to display the error message associated with this return status value.

16.4 Setting and Displaying Fields Within a Status Value

You can use the structure STS$FIELDS to set or display fields within a status value. For example, if you wish to define application-specific message numbers using the format used by VAX/VMS, you can specify a facility-wide message number, set the STS$CUST_DEF field to '1'B, assign unique numbers to messages, and define severities for the messages.

Since the fields within this structure are defined as bit strings, and it is usually more convenient to express facility or message numbers as integers, you must use the UNSPEC built-in function to convert integer values to the appropriate bit-string representation. The following example shows how to

Chapter 17
Error Signaling and Condition Handling

The standard PL/I language provides error handling and signaling through the use of the ON, REVERT, and SIGNAL statements and several built-in functions. In VAX–11 PL/I, these statements and the functions they perform have been extended to encompass VAX-specific and program- or application-specific error signaling and condition handling. This chapter describes:

- The relationship of the VAX/VMS condition-handling facility to VAX–11 PL/I's condition-handling features
- VAX–11 PL/I extensions for condition handling
- The actions that an ON-unit can take
- The search for ON-units when a condition is signaled and the default handling performed when no ON-unit exists for a given condition

All VAX–11 PL/I run-time procedures use PL/I condition handling. The information in this chapter supplements the information on ON conditions and ON-units in the *VAX–11 PL/I Encyclopedic Reference*, by providing details that are specific to PL/I programs executed under the control of the VAX/VMS operating system.

17.1 Relationship of VAX/VMS Condition Handlers to PL/I ON-Units

In the VAX/VMS environment, an exception condition is a hardware- or software-detected condition that synchronously interrupts the execution of an image. A condition handler is a procedure that exists specifically to respond to one or more such conditions; each procedure in the program can establish a condition handler. It is usually the responsibility of each handler to determine the specific condition that was signaled, and to decide whether or not to handle it.

Most high-level languages establish condition handlers by calling the VAX–11 Run-Time Library procedure LIB$ESTABLISH. The PL/I language, however, has in the ON-unit a condition handler defined to handle a specific condition. By using the keyword condition names defined by PL/I and the extensions provided by VAX–11 PL/I, you can write ON-units to handle any possible

17.1.2 Values for ON Condition Names

For any condition that is signaled, the built-in function ONCODE returns the specific 32-bit status value that describes the condition. The low-order three bits of this value contain the severity of the condition (success, warning, error, and fatal). The severity of a condition is important only when no ON-unit exists for a condition, and default condition handling is performed by either PL/I or the system (see Section 17.4, "Search for ON-Units").

All VAX/VMS-defined conditions have symbolic names associated with them. Table 17–1 lists the PL/I keyword condition names and the global symbol names for the VAX/VMS condition values associated with them. If the ONCODE built-in function is invoked in an ON-unit for the related PL/I condition name, it returns the value of the indicated global symbol.

Table 17–1: ONCODE Values for PL/I ON Conditions

PL/I Condition	VAX/VMS Global Symbol Name[1]
ENDFILE	PLI$_ENDFILE
ENDPAGE	PLI$_ENDPAGE
ERROR	A specific status value associated with the error that caused the condition to be signaled[2]
FINISH	PLI$_FINISH
FIXEDOVERFLOW	SS$_DECOVF or SS$_INTOVF
KEY	RMS$_*name*, where *name* is one of the following specific RMS condition names that describe a key error: RMS$_RNF, RMS$_DUP, RMS$_KEY, RMS$_MRN, RMS$_REX; or, PLI$_*name*, where *name* describes a PL/I run-time error, for example PLI$_CNVERR[2]
OVERFLOW	SS$_FLTOVF
UNDEFINEDFILE	RMS$_*name*, where *name* indicates a specific status value associated with an RMS error; or, PLI$_*name*, where *name* describes a PL/I run-time error[2]
UNDERFLOW	SS$_FLTUND
VAXCONDITION	Any user-defined condition value that was signaled
ZERODIVIDE	SS$_FLTDIV

1. If a PL/I condition is explicitly specified in a SIGNAL statement, the ONCODE value corresponds to the condition message associated with the condition, for example, PLI$_UNDFILE, PLI$_KEY, and so on.
2. These names correspond to the identification fields in the run-time messages. The RMS messages are listed in the *VAX-11 Record Management Services Reference Manual*. PL/I messages are listed in Appendix B of this manual.

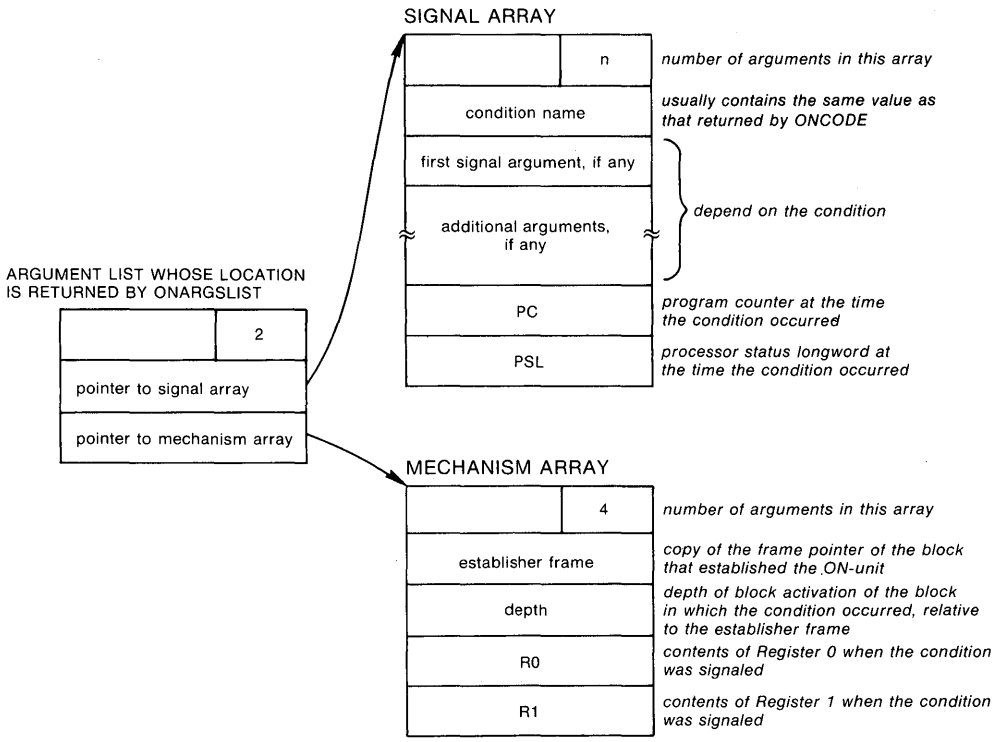


Figure 17-2: The Argument List Passed to an ON-Unit

The text module $CHFDEF contains PL/I declarations of these structures, as shown below:

```
declare chf$argptr pointer;
declare 1 chf$arglist based (chf$argptr),
    2 chf$count fixed binary(31), /* always 2 */
    2 chf$sigarglst pointer,
    2 chf$mcharglst pointer;
declare 1 chf$signal_array based (chf$sigarglst),
    2 chf$sig_args fixed binary(31),      /* argument count */
    2 chf$sig_name fixed binary(31),      /* condition name */
    2 chf$sig_arg (chf$sig_args-3) fixed binary(31),
    2 chf$pc fixed binary(31),
    2 chf$psl fixed binary(31),
  1 chf$mech_array based (chf$mcharglst),
    2 chf$mch_args fixed binary(31),      /* always 4 */
    2 chf$mch_frame fixed binary(31),
    2 chf$mch_depth fixed binary(31),
    2 chf$mch_savr0 fixed binary(31),
    2 chf$mch_savr1 fixed binary(31);
```

This module is in the default PL/I text library PLISYSDEF.TLB. You can include this module in a PL/I program by specifying:

```
%INCLUDE $CHFDEF;
```

The PL/I compiler locates this module in PLISYSDEF.TLB when it compiles the source program (see Section 2.4.7,"Default System INCLUDE Library").

17.2 VAX–11 PL/I Condition-Handling Extensions

VAX–11 PL/I defines two special keywords for the ON statement to provide additional flexibility in condition handling:

- The ANYCONDITION keyword lets you establish an ON-unit to trap all nonspecific conditions.
- The VAXCONDITION keyword lets you handle and signal either VAX/VMS-specific conditions or application-specific conditions.

These keywords, and the ways you can use them in your applications, are described individually in the next subsections.

17.2.1 An ANYCONDITION ON-Unit

An ANYCONDITION ON-unit is, in effect, a "catch-all" condition handler. It is executed whenever a condition for which no ON-unit exists is signaled in the current block or any of its descendents. Figure 17–3 illustrates three blocks in the calling sequence. Procedure A establishes an ON-unit for FIXED-OVERFLOW conditions and procedure B establishes an ANYCONDITION ON-unit. If any condition (including the FIXEDOVERFLOW condition) is signaled in procedure B after this ON statement is executed, or in procedure C, the ANYCONDITION ON-unit in procedure B is given control.

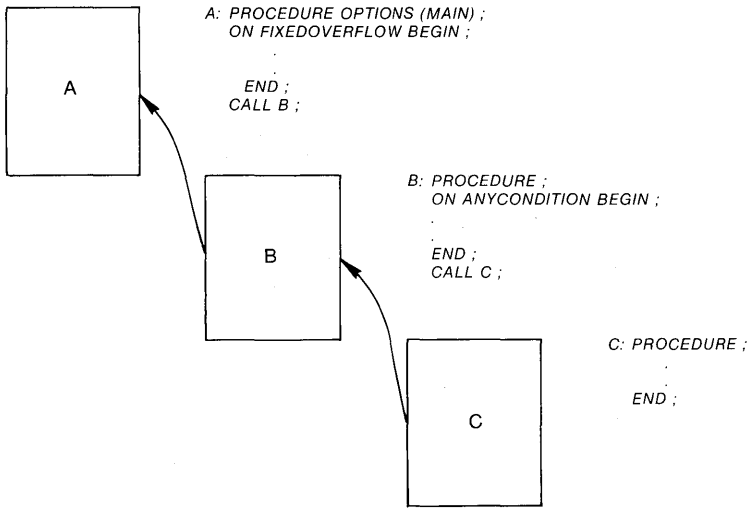


Figure 17–3: An ANYCONDITION ON-Unit

Within the VAX/VMS programming environment, the ANYCONDITION keyword provides a way to ensure that conditions signaled by PL/I procedures are not passed to non-PL/I procedures. This is particularly useful for procedures that use the VAXCONDITION condition to signal information from one block activation to another.

Note that exiting from an ANYCONDITION ON-unit with a nonlocal GOTO requires special coding. This situation is described in Section 17.3.3, "Unwind."

When these ON-units are in effect, other procedures can declare and use the names SIGNAL_STOP, SIGNAL_FOUND, and SIGNAL_NOTFOUND to signal these conditions. For example:

```
DECLARE SIGNAL_FOUND GLOBALREF FIXED BINARY;
SIGNAL VAXCONDITION(SIGNAL_FOUND);
```

Note that in this example, the application-specific values are initialized to the integers 9, 17, and 33. In actual practice, these values should be defined using the entire 32 bits of a status value, with the appropriate bit set to indicate that the value is a customer-defined value. For information on interpreting and setting status values, see Chapter 16, "Return Status Values." For information on declaring global symbols with the GLOBALDEF attribute, see Chapter 15, "Global Symbols."

17.3 Actions That an ON-Unit Can Take

The possible courses of action an ON-unit can take during its execution as a result of a condition are:

- Handle the condition and return control to the point at which the condition was signaled
- Resignal the condition and request PL/I to locate another ON-unit to handle it
- Execute a nonlocal GOTO statement and cause PL/I to unwind the call stack
- Stop the program

These courses of action are described individually.

17.3.1 Handle the Condition

A condition is assumed to be handled in PL/I when the ON-unit established for the condition completes executing without performing one of the following actions:

- Executing a nonlocal GOTO
- Calling the RESIGNAL built-in subroutine
- Signaling another condition
- Executing a STOP statement

When the condition is handled, PL/I continues execution of the program at the point of interruption.

17.3.2 Resignal the Condition

In VAX-11 PL/I, an ON-unit can decide that it does not want to handle a condition and request that, rather than returning control to the point of interruption, PL/I continue to look for another ON-unit to handle the condition.

This removal of call frames from the call stack is called an unwind. Figure 17–5 illustrates a situation in which an unwind occurs. The circled numbers indicate the order of execution. The ERROR ON-unit established in procedure A receives control when the ERROR condition is signaled in procedure C. This ON-unit executes the GOTO PRINT_MSG statement. The label PRINT_MSG is in procedure A. Thus, the call stack is unwound and the call frames for the ON-unit, procedure C, and procedure B, in that order, are removed from the stack, and execution continues at the label PRINT_MSG.

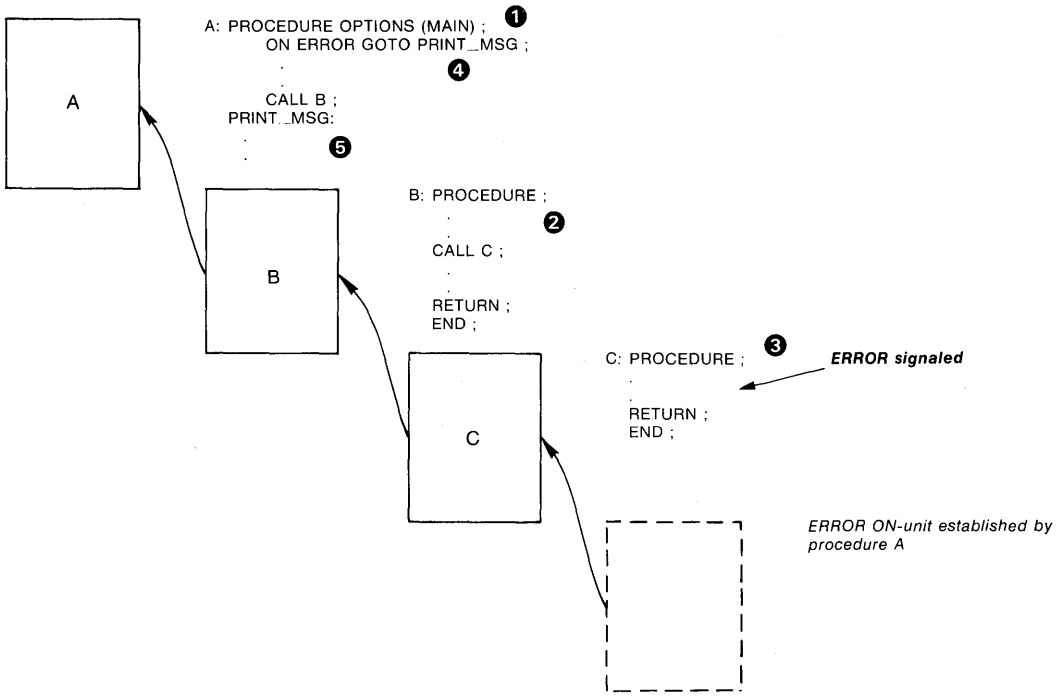


Figure 17–5: Unwinding the Call Stack

When an unwind occurs in the VAX/VMS environment, each call frame in the calling sequence is examined to determine if a condition ON-unit exists for that frame. If so, the ON-unit is called with the condition value SS$_UNWIND, and the ON-unit has the chance to perform block- or procedure-specific cleanup operations.

17.3.4 Stopping the Program

An ON-unit may specify that the program is to be terminated by executing a STOP statement. For example:

```
ON UNDEFINEDFILE(INFILE) BEGIN;
    PUT EDIT('File',ONFILE(),'undefined,  Error',ONCODE())
             (A,X,A,X,A,X,F(10));
    STOP;
    END;
```

- If the signal is the ENDPAGE condition, the default PL/I handler executes a PUT PAGE for the file, and then continues the program at the point at which ENDPAGE was signaled.
- If the signal is the ERROR condition and the severity is fatal, the default handler signals the FINISH condition. Then, one of the following occurs:
  - If a FINISH ON-unit is found, it is given a chance to execute. If it executes a nonlocal GOTO or signals another condition, program execution continues.
  - If no FINISH ON-unit is found, or if a FINISH ON-unit completes execution by handling the condition, then PL/I resignals the condition to the default VAX/VMS condition handler. This handler prints a message, displays a traceback, and terminates the program.
- If the signal is any condition other than ENDPAGE or ERROR with a fatal severity, the default PL/I handler signals the ERROR condition with the severity of the original condition. Then, one of the following occurs:
  - If an ERROR ON-unit is found, it is executed. If it completes execution by handling the condition, the program continues.
  - If an ERROR ON-unit is not found, the default PL/I handler resignals the condition. If this resignal results in control returning to the system, the default VAX/VMS condition handler prints a message and a traceback. If the error is a fatal error, the default handler terminates the program; if the error is nonfatal, the program continues.

17.4.2 Default Handling for Non-Main Procedures

If the call frame at which the procedure was entered did not specify the MAIN option, the default condition handling is as follows:

1. PL/I searches for specific ON-units in the following order:
   1. A VAXCONDITION ON-unit established for the specific condition value that is being signaled
   2. A PL/I ON-unit established for a PL/I condition name, if PL/I defines a name for the condition
   3. An ANYCONDITION ON-unit

   If one of these ON-units exists, it is executed and the search is ended. If the ON-unit completes execution by handling the condition, the program continues at the point at which the condition was signaled.
2. If no ON-units are found in any call frame, the condition is resignaled to the caller. If the resignal results in return of control to the system, the

Figure 17–7 illustrates the search sequence when a second condition occurs during the execution of an ON-unit. The circled numbers indicate the order of execution. The ERROR condition in procedure C is handled by the ON-unit established in procedure B. During the execution of this ON-unit, a FIXEDOVERFLOW condition is signaled. PL/I locates the ON-unit established for FIXEDOVERFLOW conditions in procedure C and gives it control.

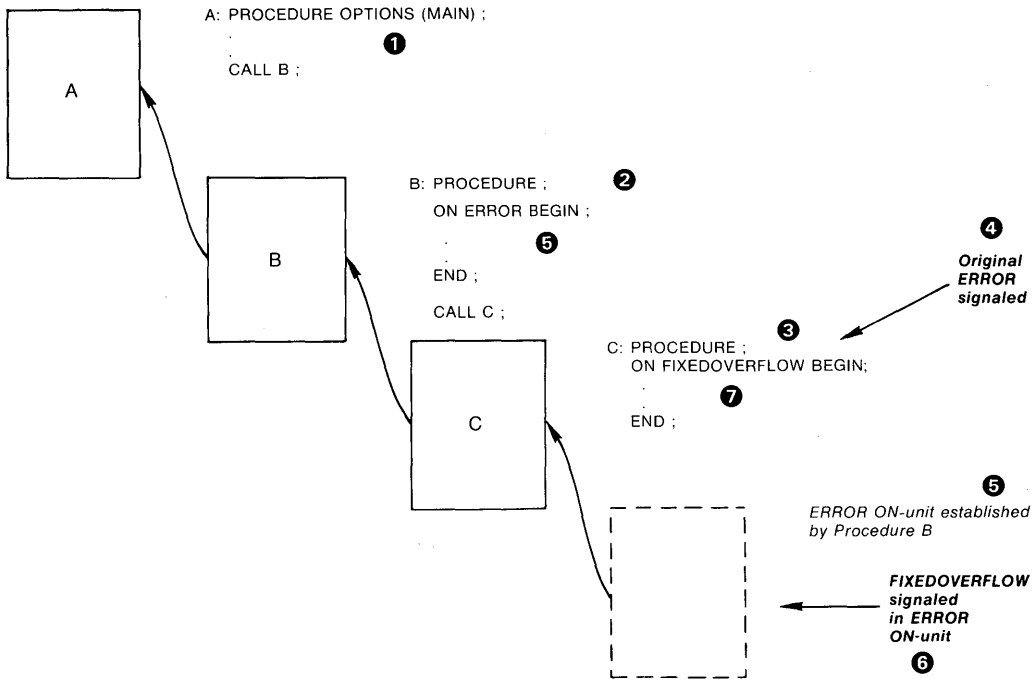


Figure 17–7: Effect of Multiple Conditions

Note that if the second condition is the same condition as the first, and the ON-unit does not establish another ON-unit, the same ON-unit will be executed repeatedly as the condition is signaled. A similar situtation results when a STOP statement is executed within a FINISH or ANYCONDITION ON-unit — that is, the program will enter an interminable loop when the STOP statement executes. The STOP statement signals FINISH, the current ON-unit is reexecuted, the STOP statement is executed again, and so on.

In a PL/I program, an ANYCONDITION ON-unit or a VAXCONDITION ON-unit established specifically to handle the SS$_UNWIND condition is invoked during the unwind. The following example illustrates a VAXCONDITION ON-unit:

```
DECLARE SS$_UNWIND GLOBALREF VALUE FIXED BINARY(31);
ON VAXCONDITION(SS$_UNWIND) BEGIN;
     CLOSE FILE(DATA_REC_TEMP) ENVIRONMENT(
                                    DELETE(NO) );
     END;
```

When an ON-unit that is handling the unwind condition completes execution, the unwind continues.

Part IV
Programming Considerations and Examples

Chapter 18
Storage Allocation and Usage

This chapter provides some general information on:

- How the compiler and linker use program sections
- How to allocate storage within an area

18.1 Program Sections

When the PL/I compiler creates an object module, it groups data in the object module into contiguous areas called program sections. The grouping is performed on the basis of the attributes of the data — for example, whether it contains executable code or read/write variables.

The compiler also writes, into each object module, information about the program sections contained in it. The linker uses this information when it binds object modules into an executable image. As the linker performs its task of allocating virtual memory for the image, it groups together program sections that have similar attributes.

18.1.1 Attributes of Program Sections

Table 18–1 lists the attributes that can be applied to program sections. The first column lists pairs of conflicting attributes.

Table 18–2: Program Sections for PL/I Variables

Storage Class Attributes	Program Section Name[1]	Program Section Attributes
EXTERNAL STATIC [2]	*name*	PIC, OVR, REL, GBL, SHR, NOEXE, RD, WRT
EXTERNAL READONLY	*name*	PIC, OVR, REL, GBL, SHR, NOEXE, RD, NOWRT
INTERNAL STATIC	$DATA	PIC, CON, REL, LCL, NOSHR, NOEXE, RD, WRT
INTERNAL READONLY	$CODE	PIC, CON, REL, LCL, SHR, EXE, RD, NOWRT
GLOBALDEF	$DATA	PIC, CON, REL, GBL, SHR, NOEXE, RD, WRT
GLOBALDEF (psect-name)	*psect-name*	PIC, CON, REL, GBL, SHR, NOEXE, RD, WRT
GLOBALDEF READONLY	$CODE or *psect-name*	PIC, CON, REL, GBL, SHR, NOEXE, RD, NOWRT

1. *name* is the identifier of the variable declared with the specified attribute. *psect-name* is the name specified in the definition of the global symbol.
2. File constants have the same attributes as EXTERNAL STATIC variables, but with the NOSHR attribute instead of the SHR attribute.

18.1.3 Sharing Program Sections with FORTRAN Procedures

In a FORTRAN program, separately compiled procedures share data by declaring COMMON sections and specifying the names of one or more variables to be placed in those sections. Each named COMMON represents a separate program section; each procedure that declares the COMMON with the same name can access the same variable.

A PL/I external variable called *name* therefore corresponds to a FORTRAN COMMON with the same name. The examples below illustrate PL/I procedures and FORTRAN procedures that share data.

STRING.PLI contains:

```
STRING: PROCEDURE OPTIONS(MAIN);
DECLARE XYZ EXTERNAL CHARACTER(20),
     PRSTRING ENTRY;
     XYZ = 'THIS IS A STRING';
     CALL PRSTRING;
STOP;
END;
```

18.2 Allocation of Storage in an Area

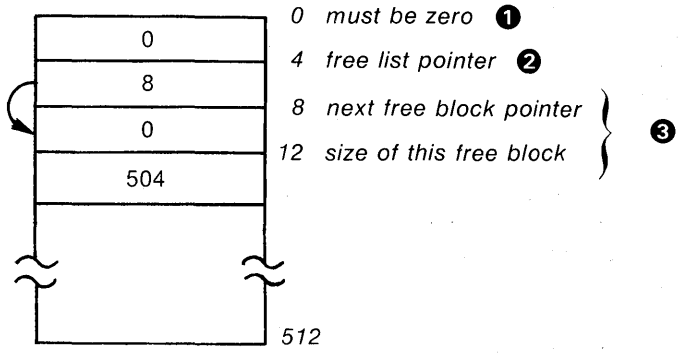
VAX-11 PL/I supports the AREA and OFFSET data type attributes, but does not provide support for allocation of storage for variables within the area. There are a variety of techniques that can provide management of space within an area; the algorithm to use for a particular application depends on the application's requirements.

This section presents an algorithm for storage management in an area that illustrates some of the considerations involved. It describes three procedures that allocate and free space within an area:

- INITIALIZE_AREA initializes an area of a given size. The area size must be a multiple of eight bytes.
- ALLOCATE_IN_AREA determines whether there is sufficient free space within the area for a given variable. If so, it returns an offset to the allocated space. If not, it returns a null offset value.
- FREE_AREA reclaims free space when a given variable is deallocated from an area.

Figures 18–1 through 18–3 illustrate calls to these procedures and the resulting area and space allocation following each call. The procedures and the algorithms used by each are described in Sample Program 18–1, which follows the figures.

```
DECLARE MAPFILE AREA(512),
        INITIALIZE_AREA ENTRY (AREA(*) );
        .
        .
CALL INITIALIZE_AREA (MAPFILE);
```



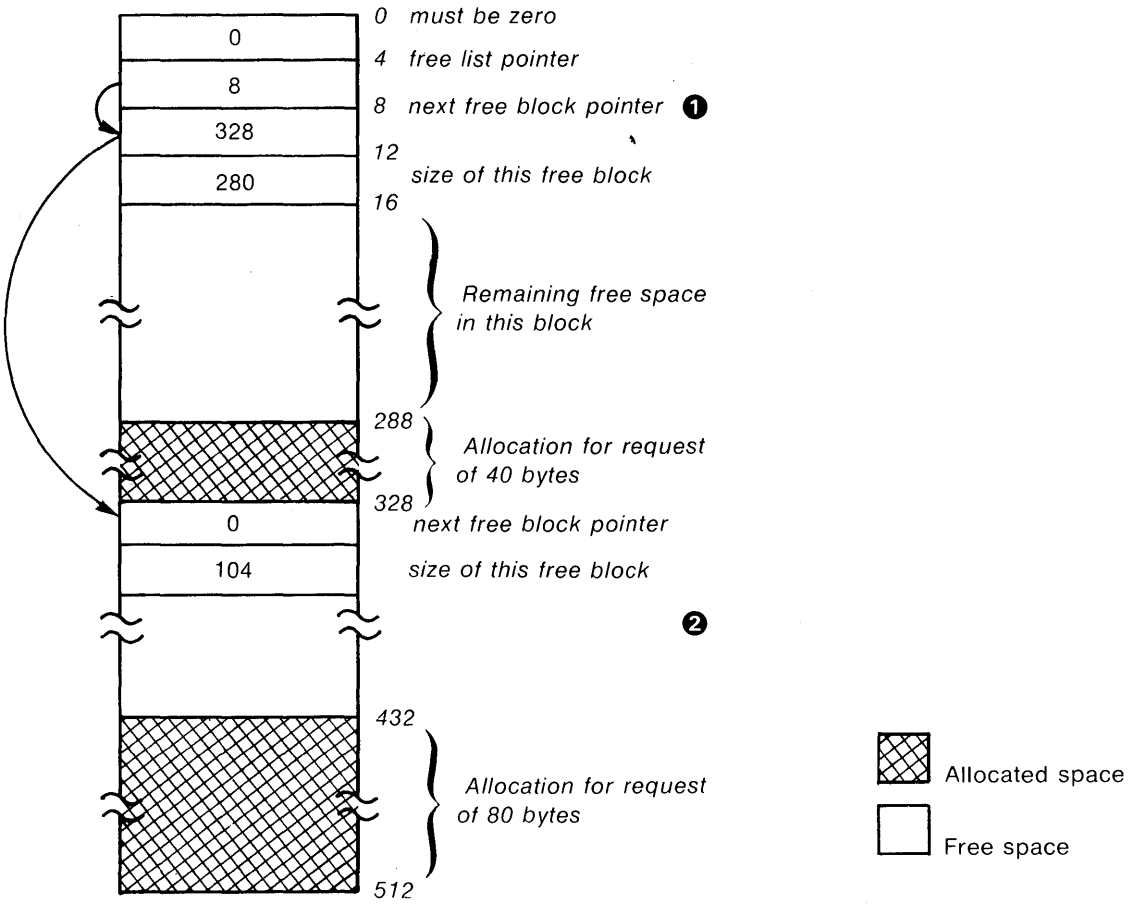
1. The first longword of an area must be zero. This longword is reserved to DIGITAL.
2. The second longword points to the beginning of the free list, that is, a chain of free blocks within the area. When this procedure initializes an area, it sets this free list pointer to point to the next longword in the block.
3. Each free block header has the format:

```
DECLARE 1 FREEBLK BASED,
          2 NEXT OFFSET (TARGET_AREA),
          2 SIZE FIXED BINARY(31);
```

   When the area is initialized, the NEXT field of the first free block header is set to a null offset value and the SIZE field is set to the size of the area, minus the eight bytes required for the reserved first longword and the free list pointer.

Figure 18–1: Initializing an Area

```
DECLARE FREE_IN_AREA ENTRY (FIXED BINARY(31), OFFSET, AREA(*));
        .
        .
        CALL FREE_IN_AREA (100, FILEOFF(2), MAPFILE);
```



1. FREE_IN_AREA updates the NEXT field of the first free block header to point to the freed space.
2. The first two longwords of the freed space are written with a free block header. The NEXT field is set to zero (since there are no more free blocks), and the SIZE field is set to indicate the number of bytes in the block.

Figure 18-3: Freeing Space Within an Area

Sample Program 18–1: Storage Management Within an Area

```
/*
This module controls allocation within an area.

There are three entry points:

        ALLOCATE_IN_AREA -      allocate space within an area
        INITIALIZE_AREA -       initialize area for storage allocation
        FREE_IN_AREA -          free storage within an area

*/                                                          ❶

ALLOCATE_IN_AREA: PROCEDURE(REQUEST_SIZE,REQUEST_AREA) RETURNS(OFFSET);

/*
        parameters
*/
DECLARE REQUEST_SIZE FIXED BINARY(31), /* request size in bytes */ ❷

        1 REQUEST_AREA,
          2 AREA_SIZE FIXED BINARY(15), /* descriptor size field */
          2 FILLER FIXED BINARY(15),    /* unused filler word */
          2 AREA_POINTER POINTER,       /* address of the area */

        TARGET_AREA AREA(AREA_SIZE) BASED(AREA_POINTER);
/*
        data structures within the input area ❸
*/
DECLARE 1 AREA_HEADER BASED(AREA_POINTER),
          2 RESERVED FIXED BINARY(31),          /* reserved to DIGITAL */
          2 FREE_LIST OFFSET(TARGET_AREA),      /* offset to first free block */
          2 LOW_LIMIT FIXED BINARY(31),         /* lowest entry */
        1 FREEBLK BASED,                /* free block entry */
          2 NEXT OFFSET(TARGET_AREA),   /* offset to next entry */ ❹
          2 SIZE FIXED BINARY(31);      /* size of entry */

/*
        local variables
*/
DECLARE NEXT_OFFSET OFFSET(TARGET_AREA),        ❺
        NEXT_OFFSET_VALUE FIXED BINARY(31) DEFINED(NEXT_OFFSET),

        FREE_OFFSET OFFSET,
        FREE_OFFSET_VALUE FIXED BIN(31) DEFINED(FREE_OFFSET),

        TEMP_OFFSET OFFSET(TARGET_AREA),        /* temporary offset variable */
        TEMP_OFFSET_VALUE FIXED BIN(31) DEFINED(TEMP_OFFSET),

        PREVIOUS_OFFSET OFFSET(TARGET_AREA),
        PREVIOUS_OFFSET_VALUE FIXED BIN(31) DEFINED(PREVIOUS_OFFSET);
```

(Continued on page 18–11)

Sample Program 18–1 (Cont.): Storage Management Within an Area

```
/*
        allocate storage in target_area

        Find first free block that is large enough to contain the requested
        allocation. Stop at the end of the list -- no space for allocation.
*/

        PREVIOUS_OFFSET = OFFSET(ADDR(FREE_LIST),TARGET_AREA);
        DO NEXT_OFFSET = FREE_LIST                                          ❻
            REPEAT(NEXT_OFFSET->FREEBLK.NEXT)
            WHILE(NEXT_OFFSET->FREEBLK.SIZE < ROUND(REQUEST_SIZE));
                    IF NEXT_OFFSET->FREEBLK.NEXT = NULL() THEN
                        RETURN(NULL()); /* no space to allocate */ ❼
                    PREVIOUS_OFFSET = NEXT_OFFSET;
                    END;
/*
        Change free block header to reflect the allocation.
        If the allocation requires the entire block, remove the header
        from the free list.

        Return offset of allocated space at end of this free block.
*/

        NEXT_OFFSET->FREEBLK.SIZE =
            NEXT_OFFSET->FREEBLK.SIZE - ROUND(REQUEST_SIZE); ❽
        IF NEXT_OFFSET->FREEBLK.SIZE = 0 THEN
            PREVIOUS_OFFSET->FREEBLK.NEXT =     ❾
                NEXT_OFFSET->FREEBLK.NEXT;
        NEXT_OFFSET_VALUE = NEXT_OFFSET_VALUE + NEXT_OFFSET->FREEBLK.SIZE; ❿
        RETURN(NEXT_OFFSET);

/*
        Initialize area header
*/

INITIALIZE_AREA: ENTRY(REQUEST_AREA);

        AREA_HEADER.RESERVED = 0;       /* DIGITAL's longword must be zero */ ⓫
        FREE_LIST = OFFSET(ADDR(LOW_LIMIT),TARGET_AREA);
        FREE_LIST->FREEBLK.NEXT = NULL();
        FREE_LIST->FREEBLK.SIZE =
            REQUEST_AREA.AREA_SIZE - 8;
        RETURN;
```

(Continued on page 18–13)

Sample Program 18–1 (Cont.): Storage Management Within an Area

```
/*
        Free an allocated block of storage in area
*/

FREE_IN_AREA: ENTRY(REQUEST_SIZE,FREE_OFFSET,REQUEST_AREA); ⓬

/*
        Search the free list to find the place to insert the freed block.
*/
        PREVIOUS_OFFSET = OFFSET(ADDR(FREE_LIST),TARGET_AREA);
        DO TEMP_OFFSET = FREE_LIST                                    ⓭
           REPEAT(TEMP_OFFSET->FREEBLK.NEXT)
           WHILE((FREE_OFFSET_VALUE < PREVIOUS_OFFSET_VALUE
               FREE_OFFSET_VALUE > TEMP_OFFSET_VALUE) &
               TEMP_OFFSET ^= NULL());
                   PREVIOUS_OFFSET = TEMP_OFFSET; ⓮
                   END;
        PREVIOUS_OFFSET->FREEBLK.NEXT = FREE_OFFSET;
        NEXT_OFFSET = FREE_OFFSET;
        NEXT_OFFSET->FREEBLK.NEXT = TEMP_OFFSET;          ⓯
        NEXT_OFFSET->FREEBLK.SIZE = ROUND(REQUEST_SIZE);
/*
        Combine any adjacent free blocks into a single free block.
*/
        TEMP_OFFSET = FREE_LIST;          ⓰
        DO WHILE(TEMP_OFFSET ^= NULL());
            NEXT_OFFSET_VALUE = TEMP_OFFSET_VALUE + TEMP_OFFSET->FREEBLK.SIZE;
            IF NEXT_OFFSET = TEMP_OFFSET->FREEBLK.NEXT THEN DO;
                TEMP_OFFSET->FREEBLK.SIZE =
                    TEMP_OFFSET->FREEBLK.SIZE + NEXT_OFFSET->FREEBLK.SIZE;
                TEMP_OFFSET->FREEBLK.NEXT = NEXT_OFFSET->FREEBLK.NEXT;
                END;
                    ELSE
                TEMP_OFFSET = TEMP_OFFSET->FREEBLK.NEXT;
            END;

        RETURN;
/*
        Subroutine to round the request sizes to 8 byte units
*/
ROUND: PROCEDURE(INPUT_SIZE) RETURNS(FIXED BIN(31)); ⓱
DECLARE (INPUT_SIZE,I) FIXED BINARY(31);
        I = MOD(INPUT_SIZE,8);
        IF I ^= 0 THEN I = 8 - I;
        RETURN(INPUT_SIZE + I);

        END ROUND;

        END ALLOCATE_IN_AREA;
```

Chapter 19
System Services

System services are procedures implemented by the VAX/VMS operating system. Although the use of some system services is restricted by privilege requirements, many services are available for general programming use. System services are described in detail in the *VAX/VMS System Services Reference Manual.*

This chapter provides information on:

- Declaring system services in PL/I
- Specifying arguments for system services
- Testing status values returned from system services

The last section of this chapter provides examples of calling system services from PL/I programs.

19.1 Declaring System Services

The default PL/I text library PLISYSDEF.TLB contains declarations for all system services as external entries. The text module names have the form:

SYS$name

where SYS$name is the name of the system service. Thus, to include the declaration of a system service you are going to use, you specify a %INCLUDE statement as in this example:

```
%INCLUDE SYS$TRNLOG;
```

The compiler, by default, locates the module SYS$TRNLOG in PLISYSDEF.TLB during its compilation.

Global symbol definitions for the entry vectors of all system services are located in the default system object module library, STARLET.OLB in SYS$LIBRARY. When you link a PL/I program, the linker searches this library by default.

Table 19–1: Input Arguments for System Services

Argument Data Type	Parameter Declaration in PLISYSDEF	Program Reference
Numeric values:	FIXED BINARY(31)	
Indicator		19–2
Channel number		19–3
Event flag		19–5
Access mode		
Binary mask		19–6
Buffer size		19–6,19–9
Process identification		
UIC		
Character strings:	CHARACTER(*)	19–1
Logical name		
Process name		19–2
Device name		
Cluster name		19–4, 19–5
Time string		
Bit masks:		19–2
32 bits or less	BIT(32) ALIGNED	
64 bits	BIT(64) ALIGNED	
Time values	BIT(64) ALIGNED	19–5
Entry mask or routine	EXTERNAL ENTRY	19–6
Buffers:	ANY	19–9
Item list		
Quota list		
AST parameter	ANY VALUE	19–6
Two-longword array	(2) FIXED BINARY(31)	

Table 19–2: Output Arguments for System Services

Argument Data Type	Parameter Declaration in PLISYSDEF	Program Reference
Numeric values:		19–1
String length	FIXED BINARY(15)	19–2
Channel number	FIXED BINARY(15)	
Access mode	FIXED BINARY(7)	
Table number	FIXED BINARY(7)	
Process identification	FIXED BINARY(31)	
Character strings:	CHARACTER(*)	
Equivalence name		19–1
Time string		
Buffers:	ANY	
I/O status block		19–6
Time value	BIT(64) ALIGNED	19–4
Two-longword array	(2) FIXED BINARY(31)	

For symbolic names that are not defined as VAX/VMS global symbols, VAX-11 PL/I provides text modules in the default INCLUDE library PLISYSDEF. All symbolic definitions in these modules are defined using %REPLACE statements. Thus, they are treated as constant identifiers rather than as variable references.

The names of the text modules, and the names and values of the symbols defined in each, are the same as the MACRO definitions in the system macro library, STARLET.MLB.

The modules required for any system service are included within the text module declaration for that service. For example, arguments to the Create Process (SYS$CREPRC) system service require symbolic names defined in the modules $PRVDEF and $PQLDEF. The module SYS$CREPRC in PLISYSDEF contains:

```
%INCLUDE $PQLDEF;       /* quota list definitions */
%INCLUDE $PRVDEF;       /* privilege bit definitions */
```

Table 19-3 summarizes the symbolic definition text modules in PLISYSDEF and gives the names of the modules in which they are included.

Table 19-3: Symbolic Definition Modules in PLISYSDEF

Module	Included in
$ACCDEF	SYS$SNDACC
$JBCMSGDEF	SYS$SNDACC, SYS$SNDSMB
$JPIDEF	SYS$GETJPI
$OPCDEF	SYS$SNDOPR
$PQLDEF	SYS$CREPRC
$PRVDEF	SYS$CREPRC, SYS$SETPRV
$PSLDEF	n/a
$SECDEF	SYS$CRMPSC, SYS$MGBLSC, SYS$DGBLSC
$SMRDEF	SYS$SNDSMB

19.4 Examples of System Services

The examples on the next few pages illustrate a number of system service calls. These examples illustrate:

- Translating a logical name
- Creating and deleting a mailbox
- Using timer and time conversion routines
- A CTRL/C routine
- Obtaining status and performance information about the current job or process

All the sample programs use the system service INCLUDE files in PLISYSDEF to declare the system services. The text of each sample program shows the INCLUDE file for the system service.

All procedures also include the module $STSDEF; however, the contents of this text module are not shown in the examples. The contents of $STSDEF are listed in this manual in Chapter 16, "Return Status Values."

Sample Program 19-1: Translating a Logical Name

```
ORION: PROCEDURE RETURNS(FIXED BINARY(31));

%INCLUDE SYS$TRNLOG;
/* Translate Logical Name system service */
declare sys$trnlog external entry (                              ❶
        char(*),                /* logical name string */
        fixed bin(15),          /* variable to receive translated length */
        char(*),                /* variable to receive translated name */
        fixed bin(7),           /* variable to receive table number */
        fixed bin(7),           /* variable to receive access mode */
        fixed bin(31) value)    /* table search disable mask */

        options(variable) returns(fixed bin(31));
%INCLUDE $STSDEF;

DECLARE CYGDES CHARACTER(6) STATIC INITIAL('CYGNUS'),

        NAMEDES CHARACTER(63),
        NAMELEN FIXED BINARY(15);                                ❷

DECLARE SS$_NOTRAN GLOBALREF FIXED BINARY(31) VALUE;             ❸

        STS$VALUE = SYS$TRNLOG(CYGDES,NAMELEN,NAMEDES,,,,);      ❹
        IF STS$VALUE = SS$_NOTRAN THEN
                PUT SKIP LIST('CYGNUS not defined');             ❺
        ELSE
            IF STS$SUCCESS THEN                                  ❻
                PUT LIST('CYGNUS is',
                         SUBSTR(NAMEDES,1,NAMELEN));
        RETURN(STS$VALUE);                                       ❼
        END;
```

Sample Program 19–2: Creating a Mailbox

```
CREATE_MAILBOX: PROCEDURE OPTIONS(MAIN) RETURNS
                        (FIXED BINARY(31));

%INCLUDE SYS$CREMBX;
/* Create Mailbox and Assign Channel system service */
declare sys$crembx external entry (                          ❶
        fixed bin(31) value,     /* permanent flag */
        fixed bin(15),           /* variable to receive channel number */
        fixed bin(31) value,     /* maximum message size */
        fixed bin(31) value,     /* buffer size */
        bit(32) aligned value,   /* protection mask */
        fixed bin(31) value,     /* access mode */
        char(*))                 /* mailbox logical name */

        options(variable) returns(fixed bin(31));
%INCLUDE $STSDEF;
%REPLACE MESSAGE_SIZE BY 132;

DECLARE PERMANENT FIXED BINARY(31) STATIC INITIAL (1), ❷
        CHANNEL FIXED BINARY(15),
        MAX_LENGTH FIXED BINARY(31) STATIC INITIAL(MESSAGE_SIZE), ❸
        PROT_MASK BIT(16) ALIGNED STATIC INITIAL('FF00'B4), ❹
        MAILBOX_NAME STATIC CHARACTER (11)
                 INITIAL('PLI_MAILBOX');


/* Call SYS$CREMBX omitting optional arguments, */
        STS$VALUE = SYS$CREMBX(
                                 PERMANENT,
                                 CHANNEL,
                                 MAX_LENGTH,      ❺
                                 ,PROT_MASK,,
                                 MAILBOX_NAME);
/*
   Return to command level with status. If SYS$CREMBX completed
   with an error, the appropriate message is displayed at the command
   level.
*/
        RETURN(STS$VALUE);
        END;
```

Sample Program 19–3: Deleting the Mailbox

```
DELETE_MAILBOX: PROCEDURE OPTIONS (MAIN) RETURNS (FIXED BINARY(31));

%INCLUDE SYS$ASSIGN;                                    ❶
/* Assign I/O Channel system service */
declare sys$assign external entry (
        char(*),                /* device name or logical name */
        fixed bin(15),          /* variable to receive channel number */
        fixed bin(31) value,    /* access mode */
        char(*))                /* associated mailbox name */

        options(variable) returns(fixed binary(31));

%INCLUDE SYS$DELMBX;
/* Delete Mailbox system service */
declare sys$delmbx external entry (
        fixed bin(31) value)    /* channel number of mailbox */

        returns (fixed bin(31));

DECLARE MAILBOX_NAME CHARACTER(11) STATIC INITIAL
                ('PLI_MAILBOX'),
        CHANNEL FIXED BINARY(15);

%INCLUDE $STSDEF;

/*
  Call SYS$ASSIGN and check return; if not successful exit
*/
        STS$VALUE = SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);
        IF ^STS$SUCCESS THEN GOTO EXIT_WITH_STATUS;     ❷

/*
 Call SYS$DELMBX and check return
*/
        STS$VALUE = SYS$DELMBX(CHANNEL);                ❸

EXIT_WITH_STATUS:
        RETURN(STS$VALUE);
        END;
```

Sample Program 19-4: Obtaining a System Time Value

```
/*
  This procedure converts a time given in ASCII format to a
  64-bit time value that is used internally by VAX/VMS.
  Input strings must be of the form:

  dd-mmm-yyyy hh:mm:ss.cc (for an absolute date or time)
  dddd hh:mm:ss.cc        (for a delta time)
*/
GETBINTIM: PROCEDURE (ASCII_STRING) RETURNS(BIT(64) ALIGNED);

%INCLUDE SYS$BINTIM;
/* Convert ASCII String to Binary Time system service */
declare sys$bintim external entry (                          ❶
        char(*),                /* ASCII string */
        bit(64) aligned)        /* variable to receive system time */

        returns (fixed binary(31));
%INCLUDE $STSDEF;

DECLARE ASCII_STRING CHARACTER(*),
        BINARY_TIME BIT(64) ALIGNED;

        STS$VALUE = SYS$BINTIM(ASCII_STRING,BINARY_TIME);    ❷
/*
   If successful, return binary time to point of invocation. Otherwise,
   return 0 - this results in absolute time 17-NOV-1858.
*/
        IF STS$SUCCESS THEN RETURN(BINARY_TIME);
        ELSE RETURN(0);
END;
```

Sample Program 19–5: Setting a Timer

```
SET_TIMER: PROCEDURE OPTIONS(MAIN) RETURNS
                        (FIXED BINARY(31));

%INCLUDE SYS$CLREF;
/* Clear Event Flag system service */
declare sys$clref external entry (
        fixed bin(31) value)    /* event flag number */

        returns (fixed bin(31));

%INCLUDE SYS$SETIMR;
/* Set Timer system service */
declare sys$setimr external entry (
        fixed bin(31) value,    /* event flag number */
        bit(64) aligned,        /* time value */
        entry value,            /* external AST procedure */
        fixed bin(31) value)    /* AST parameter */

        options(variable) returns(fixed bin(31));
%INCLUDE SYS$WAITFR;
/* Wait for Event Flag system service */
declare sys$waitfr external entry (
        fixed bin(31) value)    /* event flag */

        returns (fixed bin(31));

%INCLUDE $STSDEF;
DECLARE GETBINTIM ENTRY (
                        CHAR(*))        /* character string time */ ❶
                RETURNS (BIT(64) ALIGNED);

/* Clear event flag 5 */
        STS$VALUE = SYS$CLREF(5);
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);          ❷

/* Set the timer for 10 seconds */
        STS$VALUE = SYS$SETIMR(5,GETBINTIM('0 00:00:10'),,); ❸
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);

/* Wait for event flag 5 */
        STS$VALUE = SYS$WAITFR(5);
        IF ^STS$SUCCESS THEN RETURN (STS$VALUE); ❹

        PUT SKIP LIST('Timer up!');

RETURN(1);
END;
```

Sample Program 19-6: Establishing a CTRL/C Routine

```
SET_CTRLC: PROCEDURE RETURNS(FIXED BINARY(31));

%INCLUDE SYS$ASSIGN;                                          ❶
/* Assign I/O Channel system service */
declare sys$assign external entry (
        char(*),                /* device name or logical name */
        fixed bin(15),          /* variable to receive channel number */
        fixed bin(31) value,    /* access mode */
        char(*))                /* associated mailbox name */

        options(variable) returns(fixed binary(31));
%INCLUDE SYS$QIO;
/* Queue I/O Request system service */
declare sys$qio external entry (
        fixed bin(31) value,    /* event flag number */
        fixed bin(31) value,    /* channel number */
        fixed bin(31) value,    /* I/O function code */
        any,                    /* I/O status block */
        entry value,            /* external AST procedure */
        any value)              /* AST parameter and I/O parameters */

        options(variable) returns (fixed bin(31));
%INCLUDE SYS$WAITFR;
/* Wait for Event Flag system service */
declare sys$waitfr external entry (
        fixed bin(31) value)    /* event flag */

        returns (fixed bin(31));
%INCLUDE $STSDEF;

/*
   Declare input and output arguments. The C_INTERRUPT variable is an
   arbitrary value specified for the VAXCONDITION condition
*/
DECLARE TTCHAN FIXED BINARY(15),     /* terminal channel */     ❷
        (IO$_SETMODE,IO$M_CTRLCAST)  /* I/O function codes */   ❸
             FIXED BINARY(31) GLOBALREF VALUE;
```

(Continued on page 19–21)

Sample Program 19-6 (Cont.): Establishing a CTRL/C Routine

```
DECLARE 1 IOSB,
          2 VALUE FIXED (15),       /* Return status */ ❹
          2 NOT_USED(3) FIXED(15),
        C_AST ENTRY (POINTER),      /* CTRL/C AST routine */ ❺
        C_INTERRUPT FIXED BINARY(31) /* AST parameter */ ❻
                EXTERNAL STATIC INIT(555);

DECLARE IO_SUCCESS BIT(1) ALIGNED BASED(ADDR(IOSB.VALUE)); ❼
/* Call Assign I/O channel to get a terminal channel and then */
/* call Queue I/O Request to enable the terminal for CTRL/C   */

        STS$VALUE = SYS$ASSIGN ('TT',TTCHAN,,); ❽
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);

        STS$VALUE = SYS$QIO (1,TTCHAN,
                        IO$_SETMODE+IO$M_CTRLCAST,/* function */ ❾
                        IOSB,           /* I/O status block */
                        ,,              /* omit QIO AST argument */
                        C_AST,          /* AST routine for IO$_CTRLCAST */
                        ADDR(C_INTERRUPT),/* parameter for CTRL/C AST */
                        ,,,,);          /* unspecified p3 and p4 */
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);

        STS$VALUE = SYS$WAITFR(1);
        IF ^IO_SUCCESS THEN RETURN(IOSB.VALUE); ❿
        RETURN(1);

END;
```

19.4.4.3 Testing the CTRL/C Routine — The procedure TESTC, in Sample Program 19–8, tests the SET_CTRLC and C_AST routines. The techniques used in this procedure can be applied to any procedure in which you want to detect and respond to an external interrupt via CTRL/C. The following notes are keyed to Sample Program 19–8:

1. The procedure declares the external routine SET_CTRLC and the external variable C_INTERRUPT.
2. The value of C_INTERRUPT is used as a condition value for the VAXCONDITION keyword. This ON statement establishes an ON-unit to receive control when any procedure uses the VAXCONDITION keyword to signal the condition value C_INTERRUPT. In this short test example, the ON-unit merely displays a message, resets the CTRL/C handler, and continues the program. In effect, a CTRL/C handler can be much more elaborate: you may want to use it to close files, to advance processing to a labeled statement or block, and so on.

   Note that once a CTRL/C handler has executed, it cannot be executed again unless the I/O request that establishes a handler is reexecuted. To keep a CTRL/C handler active, it is common practice to reenable the CTRL/C routine within the AST routine itself. In this example, the ON-unit reenables the CTRL/C handler by calling SET_CTRLC.
3. The procedure calls SET_CTRLC to establish the CTRL/C handler.
4. This procedure places itself in a loop. Each time CTRL/C is entered, it displays its message for the C_INTERRUPT ON-unit and continues.

Note that when this program is run, it can be interrupted at the terminal and stopped only by the CTRL/Y function.

Sample Program 19–8: Testing the CTRL/C Routine

```
TESTC: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BIN(31));

%INCLUDE $STSDEF;
DECLARE SET_CTRLC ENTRY RETURNS(FIXED BIN(31)),
        C_INTERRUPT FIXED BINARY EXTERNAL STATIC INIT(555); ❶

ON VAXCONDITION(C_INTERRUPT) BEGIN; ❷

    PUT SKIP LIST('CTRL/C interrupt');
    STS$VALUE = SET_CTRLC();
    END;

        STS$VALUE = SET_CTRLC(); ❸
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);

        DO WHILE (1<2);
            END;          ❹
END;
```

Sample Program 19-9: TIMRE and TIMRB

```
TIME:   PROCEDURE;


%INCLUDE SYS$GETJPI;
/* Get Job/Process Information system service */
declare sys$getjpi external entry (
        fixed bin(31) value,    /* event flag number */
        fixed bin(31),          /* process identification */
        char(*),                /* process name string */
        any,                    /* item list */ (1)
        any,                    /* I/O status block */
        entry value,            /* external AST procedure */
        any value)              /* AST procedure argument */

        options (variable) returns (fixed bin(31));

/* INCLUDE file for definitions required by SYS$GETJPI */

%include $JPIDEF;       /* item codes */ (2)

%INCLUDE $STSDEF;

DECLARE 1 JPI_LIST STATIC EXTERNAL, (3)
          2 JPI_BUFIO,              /* Buffered I/O count */
            3 LENGTH FIXED BIN(15) INIT (4),
            3 CODE FIXED BIN(15) INIT (JPI$_BUFIO), (4)
            3 ADDRESS PTR,
            3 RETURN_LENGTH FIXED BIN(31) INIT (0),
          2 JPI_CPUTIM,             /* CPU time           */
            3 LENGTH FIXED BIN(15) INIT (4),
            3 CODE FIXED BIN(15) INIT (JPI$_CPUTIM), (4)
            3 ADDRESS PTR,
            3 RETURN_LENGTH FIXED BIN(31) INIT (0),
          2 JPI_DIRIO,              /* Direct I/O count     */
            3 LENGTH FIXED BIN(15) INIT (4) INIT (JPI$_DIRIO), (4)
            3 CODE FIXED BIN(15),
            3 ADDRESS PTR,
            3 RETURN_LENGTH FIXED BIN(31) INIT (0),
          2 JPI_PAGEFLTS,           /* Page faults          */
            3 LENGTH FIXED BIN(15) INIT (4),
            3 CODE FIXED BIN(15) INIT (JPI$_PAGEFLTS), (4)
            3 ADDRESS PTR,
            3 RETURN_LENGTH FIXED BIN(31) INIT (0),
          2 ENDLIST FIXED BIN(31) INIT (0);

DECLARE (T0,CLOCK_TIME) FLOAT BIN(24) STATIC EXTERNAL; (5)
DECLARE FOR$SECNDS ENTRY (FLOAT BIN(24)) RETURNS (FLOAT BIN(24));

DECLARE (BUFIO,END_BUFIO,CPUTIM,END_CPUTIM,DIRIO,END_DIRIO,
        PAGEFLTS,END_PAGEFLTS) FIXED BIN(31) STATIC EXTERNAL;
DECLARE CPUSECONDS FLOAT BIN(24);
DECLARE SS$_NORMAL FIXED BIN(31) GLOBALREF VALUE;
```

(Continued on page 19-27)

Sample Program 19-9 (Cont.): TIMRE and TIMRB

```
TIMRE:  ENTRY;
        JPI_BUFIO.ADDRESS=ADDR(END_BUFIO);      ❻
        JPI_CPUTIM.ADDRESS=ADDR(END_CPUTIM);
        JPI_DIRIO.ADDRESS=ADDR(END_DIRIO);
        JPI_PAGEFLTS.ADDRESS=ADDR(END_PAGEFLTS);

        IF SYS$GETJPI(,,,JPI_LIST,,,)^=SS$_NORMAL       ❼
                THEN PUT SKIP LIST ('Error from SYS$GETJPI');

        CLOCK_TIME=FOR$SECNDS(T0);
        CPUSECONDS=(END_CPUTIM-CPUTIM)/100E0;
        BUFIO=END_BUFIO-BUFIO;                  ❾
        DIRIO=END_DIRIO-DIRIO;
        PAGEFLTS=END_PAGEFLTS-PAGEFLTS;
        PUT SKIP EDIT ('Times in seconds','Page','Direct','Buffered')
                (A(20),A(10),A(10),A(10));
        PUT SKIP EDIT ('CPU','Elapsed','Faults','I/O','I/O')
                (A(10),A(10),A(10),A(10),A(10));
        PUT SKIP EDIT (CPUSECONDS,CLOCK_TIME,PAGEFLTS,DIRIO,BUFIO)
                (F(7,1),COLUMN(11),F(9,1),COLUMN(21),F(7,0),COLUMN(31),
                F(7,0),COLUMN(41),F(7,0));

/* After calling TIMRE, fall through here to re-initialize */

TIMRB:  ENTRY;                  ❽
        T0=FOR$SECNDS(0E0);
        JPI_BUFIO.ADDRESS=ADDR(BUFIO);          ❻
        JPI_CPUTIM.ADDRESS=ADDR(CPUTIM);
        JPI_DIRIO.ADDRESS=ADDR(DIRIO);
        JPI_PAGEFLTS.ADDRESS=ADDR(PAGEFLTS);

        IF SYS$GETJPI(,,,JPI_LIST,,,)^=SS$_NORMAL
                THEN PUT SKIP LIST ('Error from SYS$GETJPI');
        RETURN;
END;
```

Chapter 20
Mailboxes

A mailbox is a virtual input/output device that provides a means of communication for images executing in different processes. Mailboxes are used by the operating system to initiate and record system operations; they can also provide communication facilities for user applications.

This chapter provides:

- Some general information on using mailboxes
- Examples of simple procedures that perform input and output to mailboxes

Note that this chapter provides only information that is pertinent to actual input/output to mailboxes and does not describe mailbox creation. There is a system procedure to create a mailbox, the Create Mailbox and Assign Channel system service (SYS$CREMBX). For an annotated example of a call to this system service, see Section 19.4.2, "Mailbox Services."

20.1 Using Mailboxes

The next few subsections provide information on how the system controls the creation and use of mailboxes and a typical use of mailboxes in an application.

20.1.1 System Information

When a program creates a mailbox, the operating system allocates dynamic memory to store control information about the device and to buffer input and output data. The ability to create mailboxes is controlled by two separate privileges:

- The privilege to create temporary mailboxes (TMPMBX) permits a user to create a mailbox that is automatically deleted when the image that created it completes execution.
- The privilege to create permanent mailboxes (PRMMBX) permits a user to create a mailbox that continues to reside in system memory until it is specifically deleted.

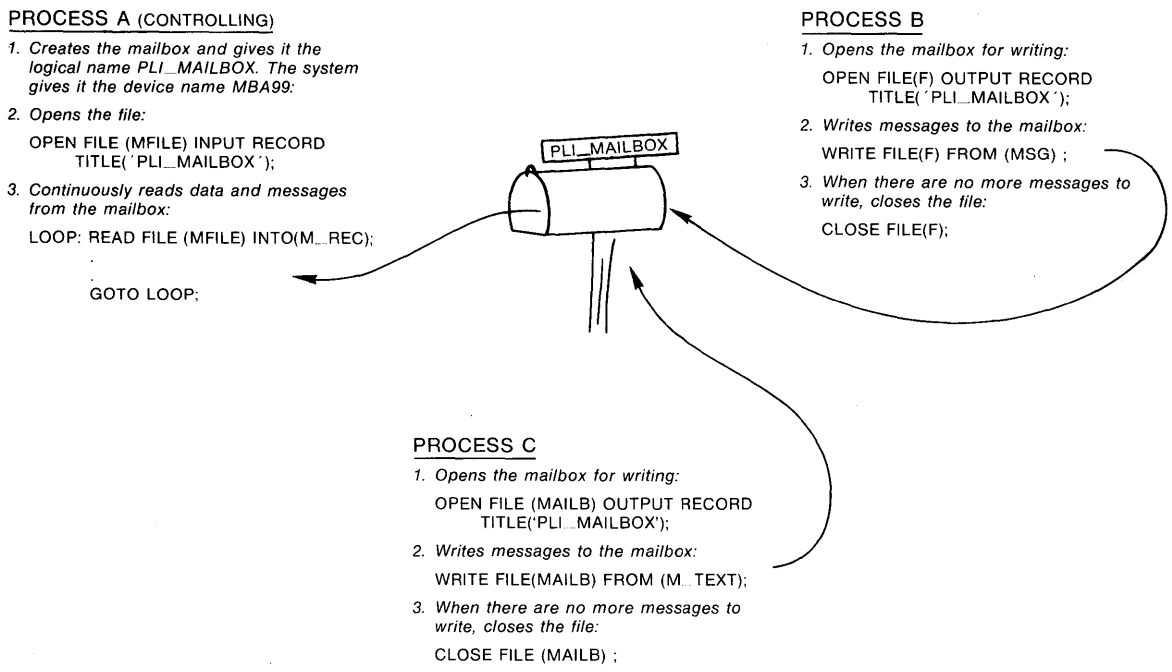


Figure 20–1: Using Mailboxes

20.1.3 Effects of the OPEN Statement

When the TITLE option of an OPEN statement specifies the logical name of a mailbox, the run-time system associates a PL/I file with the mailbox device. The OPEN statement actually assigns an I/O channel to the mailbox; a channel is an input/output path used by the operating system to perform data transfers.

Every OPEN statement executed for the same mailbox assigns another channel to the device. The system keeps a count of all channels assigned to a mailbox so it knows when to delete the mailbox.

20.1.4 Effects of the CLOSE Statement

A CLOSE statement for a mailbox disassociates the PL/I file from the device and deassigns the channel to the device. When the count of channels assigned to a temporary mailbox reaches zero, the system deletes the mailbox and its logical name equivalence, if any. When the count of channels assigned to a permanent mailbox that is marked for deletion reaches zero, the system deletes the permanent mailbox and its logical name equivalence, if any. The system service Delete Mailbox (SYS$DELMBX) must be invoked to mark a permanent mailbox for deletion.

Each time a CLOSE statement is executed for a mailbox, the file system writes an end-of-file to the mailbox. When this end-of-file is encountered during an input operation, the ENDFILE condition is signaled.

3. The OPEN statement for the mailbox specifies that it is an input file and that its logical name is PLI_MAILBOX.
4. LOGGER opens an output log file named MAILTEST.OUT.
5. This procedure establishes an ENDFILE ON-unit for the mailbox. This ON-unit transfers control to the label LOOP, which is the main read loop of the procedure. This statement ensures that LOGGER will not be accidentally terminated if an ENDFILE condition is signaled as a result of a program executing a CLOSE statement to close the mailbox file.
6. Each READ statement is followed by a test of the first field in the mailbox record. By application convention, when the value associated with the global symbol END_RUN is written to this field, it indicates that the program is complete. If this field contains any other value, LOGGER writes the record into the log file and loops to read another record.
7. When the termination value END_RUN is received, control transfers to the label FINISH, where LOGGER closes both files and returns.

Sample Program 20-1: Synchronous Mailbox Input/Output

```
LOGGER: PROCEDURE;

DECLARE (MAILFILE,OUTFILE) FILE;                    ❶
DECLARE 1 LOG_MESSAGE,
          2 TYPE FIXED BINARY(31),                  ❷
          2 SYSTEM_TIME CHARACTER(25),
          2 REQUESTOR CHARACTER(15),
          2 STATUS FIXED BINARY(31);

DECLARE END_RUN GLOBALREF FIXED BINARY(31) VALUE;

        OPEN FILE(MAILFILE) RECORD INPUT SEQUENTIAL ❸
                TITLE ('PLI_MAILBOX');
        OPEN FILE(OUTFILE) PRINT TITLE('MAILTEST.OUT'); ❹

ON ENDFILE(MAILFILE) GOTO LOOP; /* Ignore end-of-file */ ❺

LOOP:
        READ FILE(MAILFILE) INTO (LOG_MESSAGE);

        IF LOG_MESSAGE.TYPE = END_RUN THEN GOTO FINISH; ❻

        PUT FILE(OUTFILE) SKIP LIST(TYPE,SYSTEM_TIME,REQUESTOR,
                STATUS);
        GOTO LOOP;

FINISH:
        CLOSE FILE(MAILFILE);
        CLOSE FILE(OUTFILE);                        ❼
RETURN;
END;
```

Sample Program 20–2: Asynchronous Mailbox Input/Output

```
EMPTY_BOX: PROCEDURE OPTIONS(MAIN) RETURNS (FIXED BINARY(31));

%INCLUDE SYS$ASSIGN;
/* Assign I/O Channel system service */ ❶
declare sys$assign external entry (
        char(*),                /* device name or logical name */
        fixed bin(15),          /* variable to receive channel number */
        fixed bin(31) value,    /* access mode */
        char(*))                /* associated mailbox name */

        options(variable) returns(fixed binary(31));

%INCLUDE SYS$QIO;
/* Queue I/O Request system service */
declare sys$qio external entry (
        fixed bin(31) value,    /* event flag number */
        fixed bin(31) value,    /* channel number */
        fixed bin(31) value,    /* I/O function code */
        any,                    /* I/O status block */
        entry value,            /* external AST procedure */
        any value)              /* AST parameter and I/O parameters */

        options(variable) returns (fixed bin(31));

%INCLUDE SYS$WAITFR;
/* Wait for Event Flag system service */
declare sys$waitfr external entry (
        fixed bin(31) value)    /* event flag */

        returns (fixed bin(31));

%INCLUDE $STSDEF;

DECLARE MBXCHAN FIXED BINARY(15);

DECLARE (SS$_ENDOFFILE,IO$_READVBLK,IO$M_NOW) FIXED BINARY(31) ❷
        GLOBALREF VALUE,
        1 IO_STATUS,
          2 VALUE FIXED (15),
          2 BYTES_TRANSFERRED FIXED(15),
          2 NOT_USED FIXED(31),
         IO_SUCCESS BIT(1) ALIGNED BASED(ADDR(IO_STATUS.VALUE));


DECLARE MESSAGE CHARACTER(132); ❸

        STS$VALUE = SYS$ASSIGN('PLI_MAILBOX',MBXCHAN,,); ❹
        IF ^STS$SUCCESS THEN RETURN (STS$VALUE);

/* Use a DO-loop to read the mailbox; each QIO is followed
   by a test of the return status from QIO, then a wait for
   the I/O completion. Then, the status value in the I/O
   status block is checked. If it contains SS$_ENDOFFILE,
   return STS$SUCCESS. Otherwise, return error value
*/
    DO WHILE (IO_STATUS.VALUE ^= SS$_ENDOFFILE);
        STS$VALUE = SYS$QIO (1,MBXCHAN, ❺
                             IO$_READVBLK+IO$M_NOW, ❻
                             IO_STATUS,,, ❼                    ❽
                             ADDR(MESSAGE),LENGTH(MESSAGE),,,,);
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
```

(Continued on next page)

Chapter 21
Accessing Files on a Network

If your system supports DECnet–VAX facilities, and your computer is one of the nodes in a DECnet–VAX network, you can communicate with other nodes in the network by means of standard PL/I input/output statements. These statements provide two distinct types of network operations:

- Remote file access lets you read and write files on a remote node as if the file were on your local system.
- Task-to-task communication lets you exchange data directly with a job that is executing at a remote location.

Examples of both remote file access and task-to-task communication using PL/I statements are given in this chapter. For complete details on using the DECnet–VAX facilities, see the *DECnet–VAX User's Guide*.

21.1 Remote File Access

To access a file on a remote system, you include the node name in the file specification of the external file you identify for the execution of the program. For example:

```
BOSTON::DBA0:[MALCOLM]TEMPS.TST
```

This file specification identifies the file TEMPS.TST in the directory DBA0:[MALCOLM] on the node BOSTON.

You can specify a node name in a file specification in either of the following contexts:

- In the file specification in the TITLE option of an OPEN statement
- In the equivalence name you assign to a logical name before running a program that refers to a file by logical name

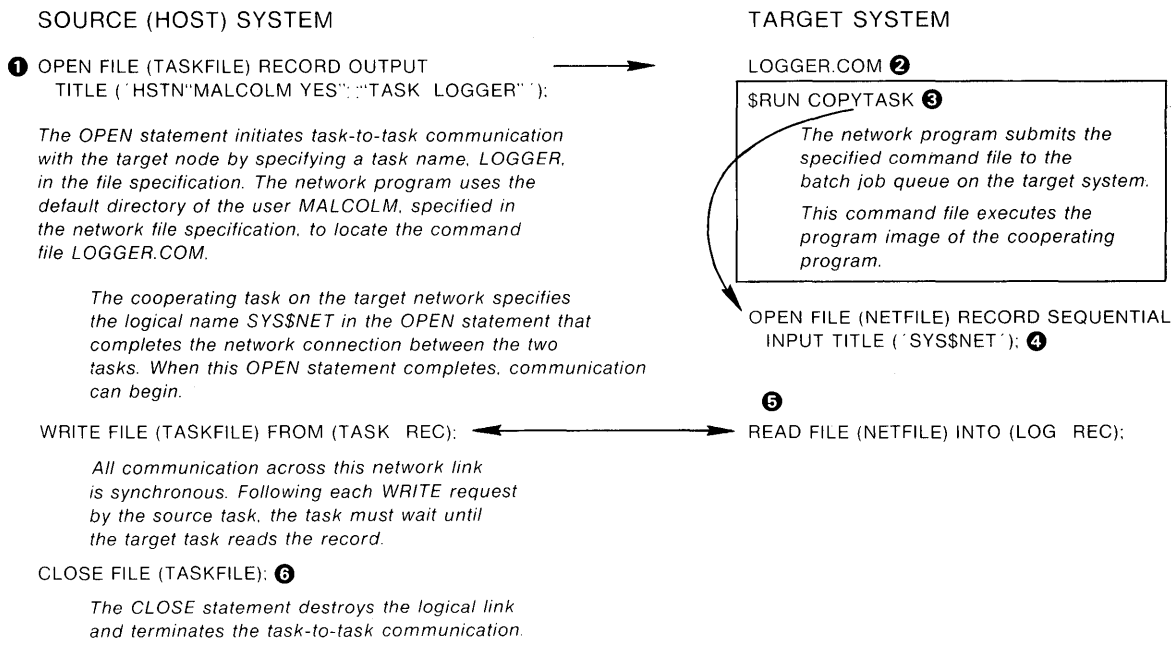


Figure 21–1: Network Task-to-Task Communication

The following notes are keyed to Figure 21–1:

1. The program that initiates the communication is called the source task. It requests a network connection to a target task by specifying a task name in a file specification that contains a node name. This OPEN statement initiates the request and associates a PL/I file with a network logical link created by DECnet.
2. DECnet locates the command file LOGGER.COM on the remote node specified in the OPEN statement. The name of the command file is specified by the task specification string, TASK=LOGGER. DECnet submits this command file for execution on the remote system.
3. The command file must contain the command necessary to initiate the execution of the cooperating program, COPYTASK.
4. The cooperating target task must complete the connection to the source task by executing an OPEN statement to open the file SYS$NET. SYS$NET is a logical name assigned by DECnet to the network job that identifies the source task's node and process.
5. After the logical link is established, the cooperating programs, or tasks, read and write data using the PL/I files associated with the logical link.
6. When either program executes a CLOSE statement for the file, the logical link is broken and an end-of-file record is written to the cooperating task.

The following notes are keyed to Sample Program 21–2:

1. The image file TARGET.EXE contains the compiled and linked code for the procedure TARGET_TASK. The declarations in the procedure TARGET_TASK include the files INFILE and OUTFILE, a structure into which messages will be read across the logical link, and a message field from which data will be written to the output file.
2. The procedure establishes an UNDEFINEDFILE ON-unit for any error conditions that occur creating the logical link; at the label FILE_ERROR, the status code is reported and the procedure exits.
3. The ENDFILE condition provides for a normal termination of the logical link. When the program SOURCE_TASK closes the file TASKNAME, an end-of-file condition is returned on the next read attempted in TARGET_TASK.
4. The first OPEN statement opens the file SYS$NET; if this open is successful, the network connection is established. The second OPEN statement opens the file TASK.DAT, the output file that will be created at the target node, in the default directory for the user name BEANS.
5. The read loop in this procedure reads a message from the logical link, edits the data, and places the record in the output file.

Sample Program 21–2: A PL/I Target Task

```
TARGET_TASK: PROCEDURE; ❶

DECLARE (INFILE,OUTFILE) FILE;     /* Files */
DECLARE 1 LOG_MESSAGE,             /* Structure to read in messages */
          2 STATUS FIXED BIN(31),
          2 TEXT CHARACTER(40) VARYING;
DECLARE MESSAGE CHARACTER(80);     /* Variable to convert message */

        PUT STRING(MESSAGE) EDIT(' ') (X(80));

ON UNDEFINEDFILE(INFILE) GOTO FILE_ERROR; /* Network errors */ ❷

ON ENDFILE(INFILE) GOTO FINISH;    /* Normal completion */ ❸

        OPEN FILE (INFILE) RECORD SEQUENTIAL INPUT
                TITLE ('SYS$NET');          /* Open SYS$NET */ ❹
        OPEN FILE(OUTFILE) RECORD SEQUENTIAL OUTPUT
                TITLE('TASK.DAT');          /* Open output log file */
LOOP:
        READ FILE(INFILE) INTO (LOG_MESSAGE);      ❺
        PUT STRING(MESSAGE) EDIT(STATUS,TEXT) (F(6),X,A);
        WRITE FILE(OUTFILE) FROM (MESSAGE);
        GOTO LOOP;
FINISH:
        CLOSE FILE(INFILE);
        CLOSE FILE(OUTFILE);
        RETURN;
FILE_ERROR:
        PUT SKIP LIST('Input file error',ONCODE());
        RETURN;
END;
```

Chapter 22
Calling SORT Procedures

VAX-11 SORT is a utility program that provides a range of sorting capabilities and options. You can use the SORT program in two ways:

- At the DCL command level, you can invoke the DCL command SORT. By specifying input and output files and sorting options, you can perform sorting functions interactively from the terminal.
- In a PL/I program, you can call SORT procedures. These procedures, or subroutines, are summarized below.

For complete details on using SORT, **see** the *VAX-11 SORT User's Guide*. For additional information on invoking procedures that are written in languages other than PL/I, and for an explanation of ways of passing parameters to non-PL/I procedures and testing the return status from a procedure, **see** Part III, "Procedure Calling and Condition Handling."

22.1 Declaring SORT Procedures

The default PL/I text library PLISYSDEF.TLB contains declarations for SORT procedures as external functions. The text module names have the forms:

SOR$name

where SOR$name is the name of the SORT procedure. Thus, to declare a SORT procedure, specify a %INCLUDE statement as in this example:

```
%INCLUDE SOR$PASS_FILES;
```

The compiler, by default, locates the module SOR$PASS_FILES in PLISYSDEF.TLB during the compilation.

All SORT parameters are passed either by descriptor or by reference.

SORT procedures do not require you to specify commas for omitted trailing arguments.

All SORT procedures are declared with the attribute RETURNS (FIXED BINARY(31)). You must invoke the procedures as functions, and you may test the value returned as an indication of success or failure.

Sample Program 22-1: Sorting Files

```
SORTEM: PROCEDURE RETURNS (FIXED);

/* Include the declarations of the SORT procedures required
   for a sort using the file interface. */

%INCLUDE SOR$PASS_FILES;                                          ❶
declare sor$pass_files external entry(
        character(*),           /* input file */
        character(*),           /* output file */
        fixed binary(7),        /* output file organization */
        fixed binary(7),        /* output file record format */
        fixed binary(7),        /* output file bucket size */
        fixed binary(15),       /* output file block size */
        fixed binary(15),       /* output file maximum record size */
        fixed binary(31),       /* output file allocation */
        fixed binary(31))       /* file options */

        options(variable) returns (fixed binary(31));
%INCLUDE SOR$INIT_SORT;
declare sor$init_sort entry(
        any,                    /* sort key buffer */
        fixed binary(15),       /* longest record length */
        fixed binary(31),       /* input file size */
        fixed binary(7),        /* number of work files */
        fixed binary(7),        /* type of sort */
        fixed binary(7),        /* key size */
        entry value,            /* comparison routine */
        bit(32) aligned)        /* sort options */

        options(variable) returns(fixed binary(31));
%INCLUDE SOR$SORT_MERGE;
%INCLUDE SOR$END_SORT;
declare sor$end_sort external entry

        returns(fixed binary(31));

%INCLUDE $STSDEF;

/*
  Declare the input and output files; these are logical names
  which must be defined before the program is run. */

DECLARE INPUT_FILE CHARACTER(6) STATIC INIT('INFILE'),            ❷
        OUTPUT_FILE CHARACTER(7) STATIC INIT('OUTFILE');

/* Declare the key buffer array required to sort the first 80
   characters of any record. This 'array' is declared as a
   structure to clarify the example. An array can also be used. */

DECLARE 1 KEY_BUFFER STATIC,                                      ❸
          2 NUMBER_OF_KEYS FIXED BINARY (15) INIT(1),
          2 KEY_TYPE FIXED BINARY (15) INIT(1), /* character */
          2 KEY_ORDER FIXED BINARY (15) INIT(0), /* ascending order */
          2 START_POS FIXED BINARY(15) INIT(1),
          2 KEY_LENGTH FIXED BINARY(15) INIT(80),
        LONGEST_RECORD FIXED BINARY(15) STATIC INIT(80);

/* Declare global symbol names for RMS values to define the output file */

DECLARE (FAB$C_VAR, FAB$C_REL) GLOBALREF FIXED BINARY(31) VALUE; ❹

DECLARE RECORD_TYPE FIXED BIN(7),
        FILE_ORG FIXED BINARY(7);

        RECORD_TYPE = FAB$C_VAR;
        FILE_ORG = FAB$C_REL;
```

(Continued on next page)

Sample Program 22–2: A Record Sort

```
/*
  This program sorts the file STATE_FILE based on the field
  CAPITAL.NAME in each record.

  Logical name equivalences are required for the input file STATE_FILE
  and an output file SORTED_FILE.
*/
STATESORT: PROCEDURE OPTIONS (MAIN) RETURNS (FIXED BINARY(31));

/*
  Declare SORT routines
*/
%INCLUDE SOR$INIT_SORT;                                          ❶
declare sor$init_sort entry(
        any,                        /* sort key buffer */
        fixed binary(15),           /* longest record length */
        fixed binary(31),           /* input file size */
        fixed binary(7),            /* number of work files */
        fixed binary(7),            /* type of sort */
        fixed binary(7),            /* key size */
        entry value,                /* comparison routine */
        bit(32) aligned)            /* sort options */

        options(variable) returns(fixed binary(31));

%INCLUDE SOR$RELEASE_REC;
declare sor$release_rec external entry(
        1,                          /* descriptor for record buffer */
          2 fixed binary(15),       /* length of record */
          2 fixed binary(15),       /* filler */
          2 pointer)                /* address of record */

        returns (fixed binary(31));

%INCLUDE SOR$SORT_MERGE;
declare sor$sort_merge external entry

        returns (fixed binary(31));

%INCLUDE SOR$RETURN_REC;
declare sor$return_rec external entry(
        1,                          /* descriptor for record buffer */
          2 fixed binary(15),       /* length of record */
          2 fixed binary(15),       /* filler */
          2 pointer,                /* address of record */
        fixed binary(15))           /* return length */

        options(variable) returns(fixed binary(31));

%INCLUDE SOR$END_SORT;
declare sor$end_sort external entry

        returns(fixed binary(31));
%INCLUDE $STSDEF;
```

(Continued on page 22–7)

Sample Program 22-2: (Cont.) A Record Sort

```
DECLARE SS$_ENDOFFILE GLOBALREF FIXED BINARY(31) VALUE, ❷
        EOF BIT(1);
        EOF = '0'B;
/*
  Key buffer and data for SORT routines
*/
DECLARE 1 KEY_BUFFER STATIC,
          2 NUMBER_OF_KEYS FIXED BINARY (15) INIT(1),
          2 KEY_TYPE FIXED BINARY (15) INIT(1), /* character keys */
          2 KEY_ORDER FIXED BINARY (15) INIT(0), /* ascending order */
          2 START_POS FIXED BINARY(15) INIT(25),
          2 KEY_LENGTH FIXED BINARY(15) INIT(20), ❸

        LONGEST_RECORD FIXED BINARY(15) STATIC INIT(196), ❹
        KEYFIELD FIXED BINARY(7),       /* Code to decide sort */
        RETURN_LENGTH FIXED BINARY(15); /* Required parameter */
/* Declare a buffer to construct each record to be passed to SORT */

DECLARE 1 STATE_RECORD,          /* complete record */ ❺
          2 KEYFIELD CHARACTER(20),
          2 STATE,
            3 NAME CHARACTER (20),
            3 POPULATION FIXED BINARY(31),
            3 CAPITAL,
              4 NAME CHARACTER (20),
              4 POPULATION FIXED BINARY(31),
            3 LARGEST_CITIES(2),
              4 NAME CHARACTER(30),
              4 POPULATION FIXED BINARY(31),
            3 SYMBOLS,
              4 FLOWER CHARACTER (30),
              4 BIRD CHARACTER (30);
DECLARE 1 RECORD_DESCRIPTOR,
          2 LENGTH FIXED BINARY(15), ❻
          2 FILLER FIXED BINARY(15),
          2 ADDRESS POINTER;
/*
  Input and output files
*/
DECLARE STATE_FILE FILE INPUT RECORD SEQUENTIAL, ❼
        SORTED_FILE FILE RECORD OUTPUT SEQUENTIAL;
/*
  call SOR$INIT_SORT
*/
        STS$VALUE = SOR$INIT_SORT(KEY_BUFFER,LONGEST_RECORD);
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
```

(Continued on page 22–9)

Sample Program 22-2: (Cont.) A Record Sort

```
/*
   Enter DO-loop to read the input file STATE_FILE. Then, call
   SOR$RELEASE_REC. The record consists of the key field concatenated
   with the contents of STATE.
*/
        OPEN FILE(STATE_FILE);
        RECORD_DESCRIPTOR.LENGTH = 196;                    ❽
        RECORD_DESCRIPTOR.ADDRESS = ADDR(STATE_RECORD);

ON ENDFILE(STATE_FILE) EOF = '1'B;

        READ FILE (STATE_FILE) INTO(STATE);
        DO WHILE (^EOF);
           STATE_RECORD.KEYFIELD = CAPITAL.NAME;  ❾
            STS$VALUE = SOR$RELEASE_REC(
                RECORD_DESCRIPTOR) ;
            IF ^STS$SUCCESS THEN RETURN(STS$VALUE);
            READ FILE(STATE_FILE) INTO(STATE);
            END;
        CLOSE FILE(STATE_FILE);
PUT SKIP LIST('**** ALL RECORDS RELEASED');
/*
  Call SOR$SORT_MERGE to sort the records that were released
*/
        STS$VALUE = SOR$SORT_MERGE();
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);  ❿
/*
  Loop through the DO-group to get back each record and write it
  to the sorted output file.
*/
        STS$VALUE = 1;
        OPEN FILE(SORTED_FILE) OUTPUT;  ⓫
        RECORD_DESCRIPTOR.LENGTH = 176;
        RECORD_DESCRIPTOR.ADDRESS = ADDR(STATE);
        DO WHILE (STS$VALUE ^= SS$_ENDOFFILE);                    ⓬
            STS$VALUE = SOR$RETURN_REC(RECORD_DESCRIPTOR,RETURN_LENGTH);
            IF STS$SUCCESS THEN WRITE FILE(SORTED_FILE) FROM (STATE);
            ELSE IF ^STS$SUCCESS & (STS$VALUE ^= SS$_ENDOFFILE)
                THEN RETURN(STS$VALUE);
            END;
        CLOSE FILE (SORTED_FILE);
/*
  Call SOR$END_SORT to finish up
*/
        STS$VALUE = SOR$END_SORT();      ⓭
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE) ;
        RETURN(1);        /* successful completion */
/*
  All errors come here to return to the command level with the
  status value of the error in R0.
*/
ERROR: RETURN(STS$VALUE);
END;
```

Appendix A
Compiler Listing Format

This appendix provides sample annotated listings from the VAX-11 PL/I compiler. It illustrates:

- Effects of the options in the /ENABLE qualifier
- The machine code generated by PL/I

Figure A-1 illustrates the default listing and describes the information provided in the listing.

Figure A-2 illustrates a storage map of the same program. The VAX-11 PL/I compiler generates a storage map if you specify /ENABLE=LIST_MAP on the PLI command; it generates a cross-reference listing in the storage map if you specify /CROSS_REFERENCE.

Figure A-3 illustrates the statistic summary generated if the /ENABLE=LIST_STATISTICS qualifier is specified.

Figure A-4 illustrates a portion of a listing of a program compiled with the /MACHINE_CODE qualifier.

The following notes are keyed to Figure A–1:

1. The name of the first level-one procedure in the source program and its identification. If the main procedure did not specify OPTIONS(IDENT) the compiler uses the default identification V001.
2. The date and time of compilation, and the version of the compiler that was used to compile the program.
3. The date and time that the file containing the source program was created, and its full file specification (to a maximum of 44 characters).
4. The page number of the listing file and the page number of the source file.
5. Compiler-generated line numbers. The compiler assigns a number to each line in the source program, including comment lines and lines read from INCLUDE files.

   Note that these line numbers do not necessarily correspond to the line numbers, if any, assigned to the file by an editor that is line-number oriented.
6. The nesting level, or depth, of each statement. The outermost procedure is always level 1. Additional level numbers are assigned to statements within internal procedures, begin blocks, and DO-groups.
7. A vertical bar (¦) character indicates a line that contains only a comment.

If the program is compiled with the qualifier /ENABLE=LIST_INCLUDE, the %INCLUDE statements are followed by the contents of the INCLUDE files, with line numbers, as follows:

```
 7    1      %INCLUDE STATE;
 8    1      DECLARE 1 STATE BASED (STATE_PTR),
 9    1                2 NAME CHARACTER (20),        /* Primary Key */
10    1                2 POPULATION FIXED BINARY(31),/* 3rd alternate Key */
11    1                2 CAPITAL,
12    1                  3 NAME CHARACTER (20),
13    1                  3 POPULATION FIXED BINARY(31),
14    1                2 LARGEST_CITIES(2),
15    1                  3 NAME CHARACTER(30),
16    1                  3 POPULATION FIXED BINARY(31),
17    1                2 SYMBOLS,
18    1                  3 FLOWER CHARACTER (30),    /* secondary - 1st alternate - Key */
19    1                  3 BIRD CHARACTER (30),      /* tertiary - 2nd alternate - Key */
20    1              STATE_PTR POINTER,
21    1              STATE_FILE FILE;
22    1
```

The listing page shown in Figure A–2 illustrates the storage map page of the program listing. This page is generated if either /ENABLE=LIST_MAP or /CROSS_REFERENCE is specified on the PLI command. The notes keyed to Figure A–2 appear below it.

```
Begin Block on line 36
----------------------

Identifier Name                 Storage     Size      Line  Attributes
---------- ----                 -------     ----      ----  ----------

INPUT_FLOWER                    automatic   32 by     37    character(30), varying, unaligned

Psect Synopsis ❸
----- --------

 Psect Name                     Allocation  Attributes
 ----- ----                     ----------  ----------

$CODE                           1054 by     position-independent, relocatable, share, execute, read
$DATA                           1 by        position-independent, relocatable, read, write
SYSIN                           450 by      position-independent, overlay, relocatable, global, share, read, write
SYSPRINT                        450 by      position-independent, overlay, relocatable, global, share, read, write
STATE_FILE                      450 by      position-independent, overlay, relocatable, global, share, read, write

Procedure Definition Map ❹
--------- ---------- ---

Line    Name
----    ----

3       FLOWERS
23       BEGIN
36       BEGIN
49       BEGIN

Command Line ❺
------- ----

/LIST=STORAGE/ENAB=LIST_MAP FLOWERS+STATETXT/LIBRARY
```

Figure A-2 (Cont.): Compiler Storage Map

1. The compiler lists the names of all external entry points in the module and their attributes.
2. For each procedure in the source program, the compiler lists each declared name, giving:
   - The user-specified identifier of the name
   - The storage class to which the name belongs
   - The amount of storage allocated for the name, where:

     bi — indicates that the size is given in bits
     by — indicates that the size is given in bytes
   - The line number on which the declaration of the name appears. Note that if a declaration is continued on more than one line (for example, in a structure declaration), the line number is always the number of the line on which the DECLARE statement is terminated.
   - The data type attributes of the name. If the name represents a member of a structure, the attributes are preceded by the offset of the structure member from the base of the structure.
3. The Program Section (Psect) Synopsis lists the program sections created by the compiler and their attributes. For an explanation of program sections and their attributes, see Section 19.1.2, "Program Sections Created by PL/I."
4. The Procedure Definition Map lists each procedure and begin block in the program, giving the line number on which the block is defined.
5. The Command Line shows the PLI command string that was processed, including input files, qualifiers, and library files.

Figure A–3 illustrates the statistical summary that PL/I includes in the listing if the /ENABLE=LIST_STATISTICS qualifier is specified. The following notes are keyed to Figure A–3:

1. The compiler accumulates statistics for each phase of its operation.
2. For each phase of the compiler's operation, it lists I/O, memory, and CPU time usage statistics.

```
FLOWERS                    11-JUN-1980 15:44:16   VAX-11 PL V1.0                Page 2
V001                       11-JUN-1980 15:42:57   _DB7:[MALCOLM]FLOWERS.PLI;17     (1)

                                  +-------------------------+
                                  | Performance Indicators |
                                  +-------------------------+

Phase  ❶                  ❷ buf i/o  dir i/o  pageflt  virtmem  workset  cputim
-------                     -------  -------  -------  -------  -------  -------
Pass 1 totals                     4        3      327        0      212       96
declare totals                    0        0       30        0      212       10
Pass 2 totals                     0        0      142       16      243       63
  live analysis                   0        0       22        0      243       13
  reorder invariants              0        0       15        0      243       12
  eliminate redundancy            0        0        5        0      243       16
  eliminate assignments           0        0        0        0      243        1
optimizer totals                  0        0       80        0      243       55
allocator totals                  1        0       33        0      243       16
  generate code list              0        0      125       64      274       59
  register allocation             0        0        1        0      274        5
  peephole optimization           0        0        0        0      274        6
  branch jump resolution          0        0        2        0      274        3
  write object module             0        1       24        0      305        8
code generator totals             1        2      167       64      305       87
total compilation                 7        6      845       80      305      352

63 lines compiled
compilation rate was 1073 lines per minute
```

Figure A–3: Compiler Performance Statistics

If you specify /MACHINE_CODE when you compile a PL/I program, the compiler prints the generated assembly language code and object code in the listing. The notes keyed to the sample machine code listing in Figure A–4 appear below the figure.

```
                        52 FE AD 9E     00E6            movab   -02(fp),r2
                           53 7C        00EA            clrq    r3
                    00000000* EF 16     00EC            jsb     PLI$PUTFILE
                        FF7E CD 9F      00F2            pushab  -0082(fp)
                     00000080 8F DD     00F6            pushl   #80
                  00000000* EF 02 FB    00FC            calls   #2,PLI$ONKEY
                     52 FF7E CD B0      0103            movw    -0082(fp),r2
                     50 FF7E CD 9E      0108            movab   -0082(fp),r0
                        51 52 3C        010D            movzwl  r2,r1

FLOWERS                                                 11-JUN-1980 15:50:20   VAX-11 PL/I V1.0                 Page 2
V001                                                    11-JUN-1980 15:42:57   _DB7:*[MALCOLM]FLOWERS.PLI;17      (1)

                    00000000* EF 16     0110            jsb     PLI$PUTLISTVCHA
                      50 FF42 CF 9E     0116            movab   $CODE+5C,r0
                        51 0A 3C        011B            movzwl  #A,r1
                    00000000* EF 16     011E            jsb     PLI$PUTLISTCHAR
                    00000000* EF 17     0124            jmp     PLI$PUT_END

27    2             ELSE
28    2                 PUT SKIP LIST('Error on key',ONKEY(),'error no.',ONCODE());

                                        012A  vcg,3:
                        51 5D D0        012A            movl    fp,r1
                     03 AF 6C FA        012D            calls   (ap),vcg,4
                         0080 31        0131            brw     vcg,5
                                        0134  vcg,4:
                          C87C          0134            .entry  vcg,code,^m<dv,iv,r2,r3,r4,r5,r6,r11>
                   5E FEF8 CE 9E        0136            movab   -0108(sp),sp
                5C 00000000 EF 9E       013B            movab   SYSPRINT,ap
                           50 7C        0142            clrq    r0
                     FE AD 01 B0        0144            movw    #1,-02(fp)
                     52 FE AD 9E        0148            movab   -02(fp),r2
                           53 7C        014C            clrq    r3
                    00000000* EF 16     014E            jsb     PLI$PUTFILE
                      50 FEF8 CF 9E     0154            movab   $CODE+50,r0
                        51 0C 3C        0159            movzwl  #C,r1
                    00000000* EF 16     015C            jsb     PLI$PUTLISTCHAR
                        FF7E CD 9F      0162            pushab  -0082(fp)
                     00000080 8F DD     0166            pushl   #80
```

Figure A–4 (Cont.): Machine Code Listing

1. The machine language code is generated in line with the PL/I source statements. Thus, you can see the code that is generated by each statement following the statement itself.

2. The listing shows, in hexadecimal, the object module location of each generated statement directly to the left of the machine language code. To the left of the object location is the object code generated by the VAX-11 PL/I compiler.

Appendix B
PL/I Messages

This appendix describes the messages produced:

- By the VAX-11 PL/I Compiler
- By the VAX-11 Run-Time System

For a description of the format of messages produced by the compiler, see Section 2.3, "Compiler Diagnostic Messages and Warning Conditions." For a description of the format and information provided by run-time messages, see Section 4.1.2, "Run-Time Errors."

B.1 Compiler Messages

The diagnostic messages produced by the VAX-11 PL/I compiler are listed below, alphabetized by identification. Within the text of a message, an item in *italics* represents data that is supplied by the compiler to provide specific information about the error.

The description of each message gives the severity, followed by additional explanatory text and suggested action. For example:

Warning. A division operation contains a fixed-point ...

Here, "Warning" indicates that the message has a severity level of Warning.

Compiler messages with severities of error or fatal require that you recompile the program after correcting the source text.

Note that the VAX-11 PL/I compiler also writes messages whose text it shares with other VAX/VMS programs; these messages are almost always self-explanatory and are not listed in this appendix; see the *VAX/VMS System Messages and Recovery Procedures* manual for descriptions of these messages.

ARITHSYN Invalid syntax in an arithmetic constant.

Error. The statement contains an arithmetic constant that is incorrectly specified.

User Action. This message may be followed by additional messages that provide syntactic reasons for the failure. Determine the type of constant required in the statement and the correct way to specify the constant. Correct the statement.

ARRAYBYVAL An argument corresponding to an array, structure, or area is not a reference. The procedure/function is "*name*".

Error. PL/I can pass aggregates and areas only by reference; thus, this error occurs when the parameter descriptor for an external entry or the parameter declaration for an internal procedure specifies a structure, dimension, or area that does not match the argument specified.

User Action. Correct either the parameter descriptor or the declaration of the argument.

ARRAYOVFL FIXEDOVERFLOW occurred in calculating the multipliers or virtual origin of the array "*array-name*".

Error. In an array with constant bounds (for some or all of its dimensions), the FIXEDOVERFLOW condition occurred when the compiler tried to calculate the multipliers and virtual origins of the array.

User Action. Check that the values specified for the array bounds are correct. Avoid using lower bound values that are very large numbers.

ASSIGNCVT Implicit conversion in an assignment, *expression* has been converted to a *data-type* target.

Warning. The data type of the indicated expression does not match the data type of the target variable in the assignment and the PL/I compiler has converted the expression to the data type of the target variable. This situation may or may not constitute an error.

User Action. To avoid this message in circumstances in which you want the compiler to convert the expression to the data type of the target, use an explicit conversion built-in function (for example, CHARACTER, BINARY, or FLOAT). You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

ATTRNOTSPC Incomplete attributes have been specified for "*variable-name*".

Error. Something is missing in a declaration.

User Action. Correct the declaration.

BADAGGARG The argument "*argument*" does not match the corresponding array, structure, or area parameter as is required by the rules for passing such arguments by reference.

Error. An array dimension or a structure declaration specified in a parameter descriptor does not match the corresponding dimension or structure of the variable specified in the procedure reference. For example, this error occurs if a parameter descriptor specifies a two-dimensional array and the procedure reference specifies the corresponding argument with a reference to a three-dimensional array.

User Action. Determine whether the parameter descriptor correctly specifies the data type, dimensions, and structure of the required parameter. If so, correct the declaration of the corresponding argument or the corresponding argument reference. If the argument is specified correctly, correct the parameter descriptor. If the procedure is a non-PL/I procedure, use the ANY attribute in the parameter descriptor.

BADANYARG A procedure argument, *expression*, is not valid for passing to the corresponding parameter, which was declared as ANY or ANY VALUE.

Error. A parameter descriptor specifies ANY or ANY VALUE, but the argument list specifies an expression that is not valid for these argument passing attributes. For example, this error occurs when an expression whose value cannot be contained within 32 bits is specified for a parameter declared with the VALUE attribute.

User Action. Determine how the argument is to be passed and correct either the parameter descriptor or the argument reference.

BADCLATTR "*name*" is declared with duplicate or conflicting attributes.
"*attribute*" conflicts with "*attribute*".

Error. This error occurs when conflicting attributes of any type are specified. Some examples of errors that produce this message are:

- When file description attributes are specified with data type attributes, or are specified for file variables or file parameters.
- When the VALUE attribute is specified for any variable for which no data type attributes are specified or is specified with the READONLY attribute.

User Action. Determine which are the correct attributes of the name and remove the invalid attribute from the declaration.

BADCLSLABL The closure label in this statement does not match the label prefix of the containing DO, BEGIN, or PROCEDURE block.

Error. Multiple closure is not permitted in VAX-11 PL/I. Each DO, BEGIN, and PROCEDURE statement in the program must have a corresponding END statement.

User Action. Verify the label on the END statement in error. The label must match the label on the most recent DO, BEGIN, or PROCEDURE statement that does not already have a corresponding END statement.

BADCODE Invalid code generation sequence.

Fatal. An internal compiler error occurred.

User Action. Gather as much information as you can about the conditions in effect when the error occurred and submit an SPR.

BADCOMPARE Invalid comparison. The operands of relational operators must both be arithmetic values, string values or compatible noncomputational items. Noncomputational data can only be compared for equality.

Error. A variable or value of a noncomputational data type is specified in a relational operation using the < or > operators or forms of these operators. For example, this error occurs if you use pointer or file variables in a comparison other than equality or inequality.

User Action. Verify that the correct variable references are specified in the expression and that the statement does not violate the rules for operands of relational operators. Correct the statement.

BADCONARG The first argument, *expression*, of a conversion built-in function is not a computational value.

Error. The indicated argument reference or expression does not have a computational data type and therefore cannot be converted to the computational data type result of the function.

User Action. Correct the argument list for the function.

BADENVAL Invalid argument in an ENVIRONMENT option.
A *value-type* was not found where expected.

Error. An ENVIRONMENT option requires either a restricted integer expression, a Boolean expression, a character string, or a variable reference. The statement in error contains an ENVIRONMENT option that specifies a value that is not consistent with its type. For example, this error occurs if a character-string argument is specified for the MAXIMUM_RECORD_SIZE argument.

User Action. Determine the data type required and correct the ENVIRONMENT option.

BADPSECT The psect specified in this statement has conflicting READONLY attributes with another definition of the same psect.

Warning. A variable declared with the GLOBALDEF attribute and with a program section name specifies the same name for the program section as another global symbol. However, one declaration contains the READONLY attribute and the other does not.

User Action. Determine whether the progam section can be declared with the READONLY attribute. If so, specify READONLY in all declarations that specify this program section. Otherwise, place read-only and read/write global symbols in different program sections.

BADRETVAL The value, *expression*, in a return statement is not valid for conversion to the *data-type* function type.

Error. The indicated return value specified in the RETURN statement does not have a data type that is valid for conversion to the data type specified in the corresponding returns descriptor.

User Action. Determine the data type that is to be returned by the function and correct either the returns descriptor or the RETURN statement.

BADSTRDCL "*name*" is an apparent structure member, but does not immediately follow a variable with a level number.

Error. A structure is incorrectly declared, or a variable declaration is preceded with an extraneous integer.

User Action. If the variable is a member of a structure, verify that the structure declaration is properly numbered and properly punctuated. The first level number in a structure declaration must be 1. If the variable is not a member of a structure, check the syntax of the declaration; remove the number preceding the variable name.

BADTARGET A reference in an assignment context is not valid for assignment.

Error. The target of an assignment is a reference to a named constant, or to a variable with the READONLY or VALUE attribute.

User Action. Correct either the reference or the declaration of the name.

BADTEXTEND Invalid end of text. Check for unbalanced apostrophes or unbalanced comments.

Error. The compiler reached the end of the input file while it was reading a character-string constant or a comment.

User Action. Locate the unterminated comment or string constant and correct it.

BADUNSPREF The argument of UNSPEC must be a reference to a scalar variable or a reference to an element of an array of scalar values.

Error. The UNSPEC built-in function is used incorrectly.

User Action. Correct the reference to UNSPEC. Its argument may not be a constant or a structure name.

BADVALUSE An expected *data-type* value was not found. One of the values in this statement has a data type that cannot be converted to the type required by the context in which the value is used.

Error. A noncomputational data type is specified when a computational data type is required, or vice versa. For example, this error occurs if the CHARACTER built-in function specifies a pointer or entry value for an argument.

User Action. Verify that the variable in question is correctly declared. If it is, correct the statement so that it refers to a variable of the correct data type.

BLANKGIVEN An arithmetic constant must be separated from the following symbol by a delimiter. A blank delimiter has been supplied.

Warning. This message indicates a syntax error in a constant, for example, an invalid character in a floating-point number or the omission of apostrophes around a bit-string constant.

User Action. Correct the constant.

BUGCHECK Compiler bug check during *phase.* Submit an SPR with a problem description.

Fatal. An internal error occurred in the compiler.

User Action. Gather as much information as possible about the conditions in effect when the error occurred and submit an SPR.

CMPLXDOPE The dope vector required for the argument "*name*" is too complicated.

Error. A structure parameter has so many members with asterisk (*) extents that the required dope vector cannot be represented in the compiler's intermediate language.

User Action. Simplify the parameters.

CNDNAMEVAL A parenthesized name or value is not valid with the "*condition-name*" condition.

Error. Only the I/O condition names and the VAXCONDITION condition name may specify values.

User Action. Correct the ON condition name in the statement.

COMPILERR Previous errors prevent continued compilation. Correct all errors and recompile.

Fatal. The compiler cannot continue.

User Action. Correct the errors indicated in the preceding messages.

CONFLATTR Attributes declared for "*name*" conflict with factored attributes.

Error. An attribute specified for a variable within a list of factored attributes conflicts with an attribute specified in the variable declaration. For example, this error occurs if a precision or extent is specified twice and the values do not match, as in: DECLARE (X CHAR(8),Y) CHAR(10);

User Action. Determine which declaration of the attribute is valid and correct the statement.

CONPREC The precision arguments of BINARY, DECIMAL, FIXED, FLOAT, and DIVIDE built-in functions must be decimal integer constants.

Error. A nonconstant value is specified for the precision argument of one of the built-in functions listed.

User Action. Correct the argument list for the built-in function in error so that it specifies a constant value for the precision argument. Replace the variable specified for the precision argument with a decimal integer constant.

CONSTCOND A condition occurred while evaluating an expression with constant operands.

Warning. The compiler evaluated an expression at compile time which resulted in the occurrence of a PL/I condition. The most common condition that occurs is FIXEDOVERFLOW.

User Action. Try to determine what expression caused the condition. Look especially at subscripts, the second and third arguments of SUBSTR references, expressions for string sizes, and array bounds. When you locate the reference (you may want to use the debugger to help locate the reference), correct it.

User Action. Check that the reference is correctly spelled; if not, correct the spelling of the reference. If the variable is not declared, declare it with the appropriate attributes for its use.

DIVIDE The divide operator was used with FIXED BINARY operands.
The compiler transformed this to DIVIDE(x,y,31).

Warning. In full ANSI PL/I, the divide operator with FIXED BINARY operands usually yeilds a scaled result, that is, it has fractional digits. However, VAX-11 PL/I does not support scaled binary data. The compiler treated the division of fixed binary values as integer division.

User Action. Rewrite the statement using the DIVIDE built-in function instead of the division operator.

DUMMYARG A dummy argument has been created for *argument*
because it does not exactly match the *data-type* parameter.

Warning. The compiler converted the argument to the data type of the corresponding parameter, and placed the result in a dummy argument. It is passing a reference to the dummy argument rather than to the actual argument to the called procedure.

User Action. If the conversion is acceptable, and if the argument will not modified in the called procedure, you need do nothing. You can enclose the argument in parentheses to suppress the message. However, if the argument must be passed by reference so that the called procecure may modify it, correct the declaration of the argument or the parameter descriptor or parameter list for the corresponding parameter.

DUPDCL This statement contains a duplicate declaration of
"*name*".

Error. The same identifier is used in more than one declaration at the same level.

User Action. Determine which declaration of the variable specifies the correct attributes, if they are different, and change the incorrect declaration.

DUPLABL This statement contains a label prefix that has appeared
on a previous statement in the same block.

Error. Two labels in the same block specify the same user-specified identifier and constant subscript.

User Action. Correct the identifier and/or the subscript and all references to it.

DUPOPTN This statement contains duplicate options.

Error. A statement contains more than one specification of the same option, for example the LIST option is specified more than once in a PUT statement.

User Action. Determine which specification of the duplicated option is the correct one and delete the other from the statement.

DUPSIGN "*token*" contains multiple sign symbols.

Error. A picture specification contains more than one sign (+ or -) symbol.

User Action. Correct the picture so that it contains only a single sign.

EMPTYARG "*name*" has been referenced with an argument list which
is incompatible with its declaration. An empty argument list
is required to satisfy the declaration.

Error. A CALL statement or a function reference specifies an argument list for a procedure that has no parameters.

User Action. Verify the arguments required for the procedure invocation. If the parameter descriptor or parameter list does not specify any parameters, the procedure invocation must not specify any arguments. Note whether the parameter descriptor list or parameter list is in error; if so, correct it. Otherwise, correct the procedure invocation.

FLTBPREC The precision specified for "*name*" exceeds the implementation's limit of FLOAT BINARY(*precision*). The maximum precision of *precision* has been supplied.

Warning. The compiler changed the precision of the floating-point variable.

User Action. Correct the declaration of the variable so that it does not specify a precision greater than the system's maximum.

FLTDPREC The precision specified for "*name*" exceeds the implementation's limit of FLOAT DECIMAL(*precision*). The maximum precision of *precision* has been supplied.

Warning. The compiler changed the precision of the floating-point variable.

User Action. Correct the declaration of the variable so that it does not specify a precision greater than the system's maximum.

IDENTSIZE An identifier contains more than 31 characters. Only the first 31 characters will be used.

Warning. The compiler truncated a user-specified identifier that is longer than 31 characters.

User Action. Shorten the identifier to 31 characters or fewer.

IMPLBLTIN *name* has been implicitly declared as a built-in function.

Warning. The undeclared name of a built-in function with no arguments has been used without an explicit empty argument list, for example DATE was specified instead of DATE().

User Action. Declare the function or specify with the empty argument list.

INCNESTLVL %INCLUDE statements cannot be nested more than 4 levels deep.

Fatal. The compiler encountered a %INCLUDE statement when it was already at the maximum nesting level of INCLUDE files.

User Action. If the LIST_INCLUDE option was not in effect when you compiled the program, recompile the program specifying /LIST and /ENABLE=(LIST_INCLUDE). Then, examine the listing to ascertain which INCLUDE file caused the error. Correct the logical nesting of the INCLUDE files and/or modules so that the nesting level is no greater than four.

INCSYN Invalid syntax in %INCLUDE statement. The correct syntax is "%INCLUDE 'file-spec';" or "%INCLUDE text-module-name;".

Error. A %INCLUDE statement is incorrectly specified.

User Action. Examine the %INCLUDE statement. If the INCLUDE file is in an individual file, the file specification must be enclosed in apostrophes. If the INCLUDE file is in a text library module, no apostrophes must be specified and the module name must not contain any punctuation marks.

INITCVT One of the initial values specified for "*name*" cannot be converted to the type of the variable.

Error. An invalid value is specified in an INITIAL attribute.

User Action. Compare the data type of the constants specified in the INITIAL attribute list with the attributes specified for the variable. Determine which has the appropriate data type and correct the other.

User Action. Correct the declaration of the parameter so that it does not specify a storage class attribute. A parameter occupies the storage of its corresponding argument at the time of the invocation, and thus cannot be allocated storage in any other way. It also cannot be used as a label.

INVSTARUSE "*name*" is declared with an * extent but is not a parameter or a descriptor.

Error. An asterisk is specified for the length of a character string or for the dimension of an array, and the string or array is not a parameter.

User Action. Correct the declaration of the variable so that its extent is specified using a constant or a valid variable declaration.

INVSUBLABL "*name*" is a subscripted label prefix previously declared with a different data type or a different number of dimensions.

Error. A label conflicts with the declaration of a variable.

User Action. If the label prefix has the same identifier as a declared variable, change either the label or the variable and correct all references to them.

ITERFACT If an iteration factor is used with a string constant, the constant must be enclosed in parentheses. This construction means "iteration" occurrences of the constant as opposed to concatenation.

Warning. An iteration factor is specified for a string constant in an INITIAL attribute, but the iteration factor is not enclosed in parentheses. The compiler assumes that the factor is in parentheses.

User Action. Place the iteration factor in parentheses, for example: INITIAL((5)('strings')).

ITERVAL "*token*" has been declared with a variable iteration factor. An iteration factor of one has been supplied.

Error. A nonconstant iteration factor was used to initialize a static variable. Nonconstant iteration factors are valid only in the initialization of automatic variables.

User Action. Specify a constant in the iteration factor. Or, declare the variable with the AUTOMATIC attribute.

LIBERROR Error while reading library "*library-name*".

Fatal. The compiler cannot read the indicated library. Either the file is not a library, or its format has been corrupted.

User Action. Verify that the library file is specified correctly, and that it is a valid VAX/VMS text library.

LIBLOOKUP "*module-name*" was not found in any of the specified libraries.

Fatal. The compiler failed to locate the indicated module in any library specified on the PLI command, in the library specified as PLI$LIBRARY, if any, or in the default INCLUDE library.

User Action. Check the PLI command line to verify that the library containing the specified INCLUDE text module was specified and that the name of the module was spelled correctly. If the library is a default user library, determine whether it has been assigned to the logical name PLI$LIBRARY.

LOCNEED "*name*" is a based variable referenced without a locator qualifier.

Error. A variable is declared with the BASED attribute without a pointer variable and is referenced without a locator qualifier (->).

User Action. Specify a pointer variable in the declaration of the variable, or specify the current pointer reference in the statement that caused the error.

NOLABL PROCEDURE, ENTRY, and FORMAT statements must have a label.

Error. The indicated statement is not labeled.

User Action. Place a label on the statement that caused the error.

NOLISTING No listing file produced.

Informational. The compiler did not create a listing file.

User Action. None.

NOLOCNEED "*name*" is a nonbased variable referenced with a locator qualifier.

Error. A locator-qualified reference is specified for a variable that does not have the BASED attribute.

User Action. Remove the locator qualifier (->) from the reference. If you expected that the variable needed a locator qualifier, verify that the variable has the BASED attribute.

NONCONEXTN "*name*" is declared with nonconstant extents but is not an automatic, based, or defined variable.

Error. The indicated variable or descriptor for a character-string, bit-string, or array variable used a variable instead of a constant to define the extent. Variables are permitted for extents only for automatic, based, and defined variables.

User Action. Correct the declaration of the variable.

NONCONINIT "*name*" has been declared with a nonconstant initial value. Static variables must have constant initial values.

Error. A static variable is incorrectly initialized.

User Action. Correct the declaration so that it uses only constant values in the INITIAL attribute.

NORETVAL All RETURN statements in a function must return values.

Error. A RETURN statement in a function does not specify a value.

User Action. Specify a value on the RETURN statement, ensuring that the data type of the value matches the data type specified on the RETURNS option of the PROCEDURE statement.

NOTARITH Implicit conversion. A nonarithmetic expression, *expression*, has been used in a context requiring an arithmetic value.

Warning. A bit-string or character-string expression was used in a context where an arithmetic expression is required. The PL/I compiler has converted the expression to arithmetic. This situation may or may not constitute an error.

User Action. To avoid this message in circumstances in which you want the compiler to convert the expression to the appropriate arithmetic data type, use the BINARY built-in function to convert a bit string or the BINARY, FIXED, DECIMAL, or FLOAT built-in function to convert a character string. You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

NOTARRAY The first argument to a LBOUND, HBOUND, or DIM built-in function must be an array reference.

Error. The argument list for one of the functions listed is incorrectly specified.

User Action. Correct the argument list for the built-in function.

NOTBASED The variable "*name*" is not a BASED variable.

Error. The target variable specified in the ALLOCATE statement does not have the BASED attribute.

User Action. Verify that the variable is specified correctly. If so, correct the variable's declaration so that it specifies BASED.

User Action. Verify that the reference in the condition name is to a file constant or file variable that is declared correctly.

NOTINT Implicit conversion. A noninteger expression, *expression*, has been used in a context requiring an integer value.

Warning. A character-string, bit-string, or noninteger arithmetic expression is used in a context where an integer is required. The PL/I compiler has converted the expression to an integer. This situation may or may not constitute an error.

User Action. To avoid this message in circumstances in which you want the compiler to convert the expression to an integer, use the BINARY built-in function to convert bit- or character-string expressions to an integer. You can use the FIXED built-in function to convert floating-point expressions or fixed-point decimal. expressions with a nonzero scale factor to integers. You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

NOTINTBND A constant has been used as an array bound, but it is not an integer constant whose value is less than 2**29. If a constant is used as a bound, it must be a valid integer.

Error. An invalid constant is specified for an array bound.

User Action. Verify that the bound specified is within the valid range for array bounds and correct the declaration. Note that this error may occur when any parenthesized expression follows an identifier in a declaration. In this context, the message indicates that the statement syntax is in error and must be corrected.

NOTINTCON An expected optionally signed integer was not found.

Error. A nonconstant expression is specified in a context that requires an integer constant.

User Action. Specify an integer constant.

NOTINTVAL A noninteger value is specified as a VAXCONDITION value.

Warning. A noninteger value is specified for the VAXCONDITION condition name.

User Action. Specify an integer value in the ON, SIGNAL, or REVERT statement.

NOTLOCATOR A value that is not a pointer or offset value has been used in a context requiring a locator value.

Error. The reference specified as a locator qualifier is not a pointer or offset value.

User Action. Correct the locator-qualified reference so that the item on the left of the locator qualifier (->) is a pointer or offset.

NOTSCALAR An array or structure value has been used in a context that requires a scalar value.

Error. An array or structure reference is specified in an invalid context, for example, as an operand of an arithmetic operation.

User Action. Correct the statement so that it does not contain a reference to an aggregate.

NOTSUBROUT The reference in a CALL statement is not a subroutine reference.

Error. A CALL statement specifies the name of an entry that has the RETURNS attribute.

User Action. If the invoked procedure is a function, correct the statement in error so that the procedure is invoked as a function reference. Otherwise, delete the RETURNS option from the PROCEDURE statement of the procedure so that it can be invoked by a CALL statement.

REQINIT An INITIAL attribute must be specified for "*name*".

Warning. The indicated name is not specified with the INITIAL attribute. This error occurs for names declared with the READONLY or VALUE attributes. Since names with these attributes cannot be modified, their values are unpredictable if they are not initialized.

User Action. Specify the INITIAL attribute to give the name a value.

RETANY A returns descriptor must not specify ANY as its data type.

Error. The ANY attribute is specified in a returns descriptor.

User Action. Correct the returns descriptor so that it specifies data type attributes for the return value. The ANY attribute is valid only for parameter descriptors for non-PL/I procedures.

RETLENGTH A RETURNS attribute must not specify an array, structure, or area.

Error. The data type specified in a returns descriptor is an aggregate or area.

User Action. Ensure that the returns descriptor in the RETURNS option of the PROCEDURE statement for the function does not specify an aggregate or area value.

RETSTAR Invalid *-extent in a RETURNS attribute.

Error. An asterisk is specified for an extent or precision in a RETURNS attribute. The only valid use of an asterisk in a RETURNS attribute is RETURNS (CHARACTER(*)).

User Action. Specify a value in the RETURNS attribute.

RETURNON A RETURN statement is not allowed in an ON-unit.

Error. A RETURN statement is specified within a begin block specified for an ON-unit.

User Action. To exit from the program, use a nonlocal GOTO within the ON-unit.

RETVAL A RETURN statement in a subroutine cannot return a value.

Error. A RETURN statement in a subroutine specifies a value.

User Action. Ascertain whether the indicated procedure is to be invoked as a subroutine or as a function. If it is a subroutine, remove the return value from the RETURN statement. If it is a function, specify the RETURNS option on its PROCEDURE statement.

RETVALCVT Implicit conversion of the return value, *expression*, to the function type *data-type*.

Warning. The data type specified in a RETURN statement does not match the data type given in the corresponding returns descriptor, and the compiler has performed an implicit conversion of the value to the specified data type. In the case of a procedure with multiple entry points, this message may be issued once for each occurrence of a RETURN statement that requires an implied conversion.

User Action. If the conversion is desirable, either use a specific conversion built-in function to return the value, as in RETURN(CHAR(n)). Either correct the RETURNS option on the PROCEDURE or ENTRY statement that is in error, or correct the value specified in the RETURN statement.

STRDEPTH The depth of nesting of a structure exceeds the implementation's limit of 16.

Fatal. A structure contains too many levels.

User Action. Correct the declaration of the structure. If necessary, modify the structure so that it has no more than 16 levels.

STREFCNT A structure-qualified reference contains more than 15 qualifying names.

Error. A reference in the form name1.name2.name3... contains more than 15 names.

User Action. Examine the structure-qualified reference and compare it with the declaration of the structure, to ensure that each qualifying name is specified correctly.

STRGTOOBIG The length of a name or constant exceeds the implementation's limit of 1000 characters.
Check to see if all string constants are properly delimited with ´ and that any contained ´s occur in pairs.
Also check for unbalanced /* */.

Fatal. The compiler read more than 1000 characters following the occurrence of an open apostrophe (´) or comment (/*).

User Action. Locate the beginning of the unterminated string or comment, and terminate it at the appropriate location.

STRINGBIF The argument of the STRING built-in function must be a variable that is suitable for use in string overlay defining. It must contain only bit or only character data and must not be VARYING or ALIGNED or be an unconnected array.

Error. The STRING built-in function is used incorrectly.

User Action. Verify that the correct argument is specified for the STRING built-in function. If the argument seems correct, be sure that its declaration does not violate any of the rules given in the message.

SUBRANGE The integer value "*token*" does not lie in the range *minimum* : *maximum*.

Error. The PL/I compiler detected a reference to an array in which a subscript is not within the declared bounds of the array or an argument in a SUBSTR built-in function that references a position that is not within the extent of the target string. This message is issued only if /CHECK was specified on the PLI command to compile the program.

User Action. Correct the reference to the array.

SUBROUT The subroutine *name* has been called as a function.

Error. The statement contains a reference to a procedure that does not have the RETURNS attribute.

User Action. If the invoked procedure is a subroutine, correct the statement in error so that the procedure is invoked with a CALL statement. Otherwise, delete the RETURNS attribute from the PROCEDURE statement of the procedure so that it can be invoked by a function reference.

SYMTABOVFL The total number of symbol table pages exceeds the implementation's limit. Reduce the number and size of names, constants, extent expressions, and argument or returns descriptors.

Fatal. The program is too complex.

User Action. Simplify the program.

TOOMANYERR The total number of errors exceeds the implementation's limit of 100.

Fatal. The compiler cannot continue.

User Action. Correct the errors indicated by the preceding messages.

TOOMANYOPS More than 253 operands have been used with an operator, function, or call.

Error. An expression contains more than 253 operands.

User Action. Simplify the statement in error.

TOOMANYSUB *"name"* has been referenced with too many subscripts.

Error. The number of subscripts in the reference to an array element exceeds the number of dimensions of the array.

User Action. Examine the declaration of the array to determine the number of subscripts required, determine the subscript(s) in excess, and correct the statement.

TOOMANYVAL Excess initial values have been specified for *"name"*.

Warning. An INITIAL list for an array declaration specifies more constant values than array elements. Or multiple values were specified for a scalar constant. A list of values is valid only in an array declaration. The declaration DCL (A,B) ... INIT(1,2) initializes both A and B to 1. The second value specified is ignored.

User Action. Delete the excess values.

TOPOLOGY An assignment target, parameter descriptor, or returns descriptor is not an array or structure of the proper shape to receive the array or structure value being assigned to it.

Error. An array or structure is being assigned or passed to an array or structure with a different dimension, data type, or arrangement.

User Action. Compare the declaration of the array or structure that is specified as an argument or as a source expression with the corresponding parameter or target expression. Determine which declaration is correct and modify the incorrect specification.

UABORT Compilation terminated by user.

Fatal. You interrupted the compilation with CTRL/C and the compiler has terminated.

User Action. None.

UNDCLBASE *"name"* is undeclared and has been used in an ALLOCATE statement as the name of a based variable.

Error. The target variable in the ALLOCATE statement is a name that is not declared.

User Action. Verify that the variable is specified correctly. If so, declare it with the BASED attribute.

UNDCLPARM *"name"* is an undeclared parameter. It has been declared in its containing block and will acquire default attributes.

Warning. A name specified in a parameter list is not declared with data type attributes. The compiler gave the parameter the attributes FIXED BINARY(31).

User Action. Declare the parameter with the appropriate data type attributes.

UNRSTREF A structure-qualified reference to "*name*" cannot be resolved to any declaration known to this block.

Error. A reference in the form name1.name2.name3... cannot be resolved.

User Action. Examine the structure-qualified reference in the statement that caused the error. Verify that the structure member that is referenced is a part of the specified structure. Correct the reference so that it refers to the correct structure or to the correct member.

UPPRGTRLOW One of the bounds declared for "*array-name*" is invalid because the lower bound is greater than the upper bound.

Error. An array is incorrectly declared.

User Action. Correct the declaration of the array variable in error so that all bounds are valid. In the declaration of the a bound x:y, the value of x must be numerically less than the value of y.

VALSIZE The size or precision of "*name*" is incompatible with the VALUE attribute.

Error. A parameter descriptor or variable declared with the VALUE attribute specifies a fixed binary value with a precision not equal to 31 or a bit-string value with a length not equal to 32.

User Action. Correct the declaration so that it specifies a variable that requires 32 bits or less of storage.

VALTYPE The data type of "*name*" is incompatible with the VALUE attribute.

Error. The VALUE attribute is specified for a variable that does not have either the FIXED BINARY or BIT(32) ALIGNED attribute.

User Action. Correct the declaration.

VARYING "*name*" has been declared with the VARYING attribute. Only CHARACTER variables may be declared VARYING.

Error. The VARYING attribute is specified for a variable to which it cannot be applied.

User Action. Correct the declaration so that it does not specify VARYING.

VARYSCALE The scale factor, q, specified for "*name*" is not in the range $0 \leq q \leq p$, where p is the variable's precision. The scale factor has been set to zero.

Warning. A scale is specified for a variable that is not in its declared range.

User Action. Specify a scale factor in the allowed range.

WHATBIF "*name*" is not a built-in function or procedure known to this implementation. If this is an external entry, it must be declared by a DECLARE statement with an ENTRY attribute.

Error. A reference to a procedure cannot be resolved.

User Action. Verify that the variable referenced in the statement is a valid subroutine or function. If it is an external function, declare it with the ENTRY attribute.

ZEROSCALE "*name*" has been declared with a nonzero scale factor. A zero scale factor has been supplied.

Warning. A variable is declared FIXED (p,q) or FIXED BINARY (p,q).

User Action. Either specify the DECIMAL attribute or delete the scale factor, q, from the declaration.

For other types of print files, you may want to take special action for the ENDPAGE condition and code an ON-unit to perform the action.

ERROR PL/I ERROR condition

Fatal. This message is displayed whenever the ERROR condition is signaled and not handled within the procedure.

User Action. This message is usually followed by additional messages that indicate the specific error that occurred. Examine these messages to determine the corrective action required.

FINISH PL/I Program FINISH condition

Success. This message is displayed when the FINISH condition is signaled and the program has no ON-unit for the FINISH condition.

User Action. In many cases, this message is displayed when you have interrupted a program with CTRL/C or CTRL/Y and executed another program or a DCL command. In these cases, no action is required. Otherwise, you may want to write an ON-unit to respond specifically to the FINISH condition in a program. For a description of image exit, and the circumstances under which PL/I signals the FINISH condition, see Section 4.1.1, "Image Exit."

FIXOVF PL/I FIXEDOVERFLOW condition.

Fatal. This message is displayed when the FIXEDOVERFLOW condition occurs or is signaled and no ON-unit exists for FIXEDOVERFLOW.

User Action. Determine the variable whose value overflowed and give it a larger precision, or verify that the program logic is correct and is not trying to assign a value larger than it should to the variable. If the condition is expected, code an ON-unit in your program that handles this condition.

KEY PL/I KEY condition on file *'file-spec'*

Fatal. This message is followed by one or more messages that indicate the specific error that occurred while processing the key on the given file.

User Action. Determine the specific error that occurred by examining the accompanying RMS message. Verify in your program that the correct key value was specified in the I/O statement, that the data type of the key value can be converted to the data type of the given key, and so on. Also determine whether the file to which the I/O was attempted is the correct file. If appropriate, write an ON-unit to handle the KEY condition.

UNDFILE PL/I UNDEFINEDFILE condition on file *'file-spec'*

Fatal. This message is followed by one or more messages that indicate the specific error that occurred opening the given file.

User Action. Determine the corrective action from the accompanying messages. Verify the file specification in the FILENAME message to determine whether the correct defaults are being applied, whether all required logical name assignments are in effect, and so on.

VAXCOND User-defined condition, *condition-value*

Warning. This message is displayed when VAXCONDITION is signaled and no ON-unit exists to handle the specific numeric condition value.

User Action. Verify that the condition value specified in the SIGNAL statement matches the condition value in a corresponding ON-unit. Correct the source program and recompile.

ZERODIV PL/I ZERODIVIDE condition.

Fatal. This message is displayed when the ZERODIVIDE condition occurs; that is, the divisor in a division operation has a value of zero. This message is displayed when the condition is not handled by an ON-unit within the PL/I program.

User Action. Determine the statement that caused the error and correct the program logic, if possible. If practical, code an ON-unit to detect the condition and take appropriate action.

CONFIXLEN FIXED_LENGTH_RECORDS conflicts with other attributes or options.

Informational. The file's attribute list contains the FIXED_LENGTH_RECORDS option and an option that conflicts with it.

User Action. Consult the option description(s) in Chapter 6 to determine the options in conflict and correct the program.

CONPRINTCR CARRIAGE_RETURN_FORMAT conflicts with PRINT attribute.

Informational. A PL/I file with the PRINT attribute has variable with fixed-length control records; the carriage control information is provided by PL/I. The CARRIAGE_RETURN_FORMAT option of ENVIRONMENT cannot be specified for it.

User Action. Determine whether the file is to be a PL/I PRINT file or a file with VAX/VMS carriage return format and correct the file's attribute list.

CONPRTFRM PRINTER_FORMAT conflicts with other attributes or options.

Informational. The ENVIRONMENT option PRINTER_FORMAT conflicts with the CARRIAGE_RETURN_FORMAT option and with the PRINT and STREAM file description attributes.

User Action. Correct the file's attribute list.

CREINDEX Attempting to create an indexed file. Use RMS DEFINE.

Informational. A file was opened with the OUTPUT attribute and with the ENVIRONMENT option INDEXED. You cannot create an indexed sequential file in a PL/I program. Indexed files can only be opened for UPDATE or INPUT.

User Action. Use the RMS-11 utility program DEFINE to create the file. Correct the program to open the file with the UPDATE attribute and write records to it.

CVTPICERR Error in picture conversion.

Informational. A value could not be edited as specified by the corresponding picture.

User Action. If the value is negative, be sure that the picture includes one of the sign characters.

ENDSTRING End of string encountered during GET STRING or PUT STRING.

Informational. A GET STRING statement attempted to read past the end of the source string variable; or, a PUT STRING statement attempted to write past the end of the target string variable. This error occurs most frequently when a LIST option is specified on a GET STRING statement and the target string does not have either a trailing blank or a comma.

User Action. Verify the length of the target or source string variable, the data types specified in the GET or PUT list, correct the program, and recompile.

ENVPARM PL/I compiler/run-time error. Please submit an SPR.

Informational. An error occurred in the execution of the PL/I compiler or a run-time module.

User Action. Gather as much information as possible about the circumstances under which the error occurred and submit an SPR report.

FILEIDENT PL/I compiler/run time error. Please submit an SPR.

Informational. An error occurred in the execution of the PL/I compiler or a run-time module.

User Action. Gather as much information as possible about the circumstances under which the error occurred and submit an SPR report.

FILENAME File name: *'file-spec'*

Informational. This message specifies the VAX/VMS file specification of the file to which input/output was attempted.

User Action. Examine this informational message to determine the full specification of the VAX/VMS file on which the input/output that failed was attempted. From this name, you can verify whether the file was correctly specified in the TITLE option, whether the correct logical name assignments exist, whether the correct defaults are being applied, and so on.

INVFMTPARM Invalid format parameter specified.

Informational. A value specified for a format item was not a positive integer; or, the value was not in the valid range for the given format item. For example, this error occurs if a negative number is specified for the A or B format item, or if a value greater than 31 is specified for the F format item.

User Action. Correct the value specified for the format item in the source program and recompile.

INVFORGKEY Invalid file organization for KEYED access.

Informational. The KEYED attribute was specified for a file that cannot be accessed by key, for example, a magnetic tape file.

User Action. Verify that the correct file is being opened by checking the TITLE and DEFAULT_FILE_NAME options, if any, logical name assignments, and file specification defaults. If the file is the expected file, correct the attribute list so that it does not specify the KEYED attribute.

INVFORMAT PL/I compiler/run-time error. Please submit an SPR.

Informational. An error occurred in the execution of the PL/I compiler or a run-time module.

User Action. Gather as much information as possible about the circumstances under which the error occurred and submit an SPR report.

INVFXCSIZ Invalid FIXED_CONTROL_SIZE specified.

Informational. The value specified in the FIXED_CONTROL_SIZE ENVIRONMENT option is not in the range of 0 through 255.

User Action. Verify that the expression in the FIXED_CONTROL_SIZE option is correctly specified or that it refers to the correct variable. Or, choose a fixed-control size that is within the valid range. Correct and recompile the program.

INVINDNUM Invalid INDEX_NUMBER specified.

Informational. The value specified for the INDEX_NUMBER option does not have a corresponding index in the indexed sequential file.

User Action. Verify that the expression specified in the option is correct or that it refers to the correct variable. Or, specify an index number that is in the proper range, ensuring that the indexed sequential file was defined with the correct number of index keys. Correct and recompile the program.

INVMAXREC Invalid MAXIMUM_RECORD_SIZE specified

Error. The value specified for the MAXIMUM_RECORD_SIZE option of ENVIRONMENT is not in the range of 0 through 32767.

User Action. Correct the value so that it is not larger than 32767.

INVMLTBLK Invalid MULTIBLOCK_COUNT specified.

Informational. The value specified in the MULTIBLOCK_COUNT count of the ENVIRONMENT option is not in the range of 0 through 127, or is not a valid integer expression.

User Action. Verify that the expression in the MULTIBLOCK_COUNT option is correct, or that the correct variable reference is specified. Correct and recompile the program.

INVMLTBUF Invalid MULTIBUFFER_COUNT specified.

Informational. The value specified in the MULTIBUFFER_COUNT count of the ENVIRONMENT option is not in the range of -128 through 127, or is not a valid integer expression.

User Action. Verify that the expression in the MULTIBUFFER_COUNT option is correct, or that the correct variable reference is specified. Correct and recompile the program.

LINESIZE Invalid LINESIZE specified.

Informational. The value specified in the LINESIZE option exceeds the implementation's limit of 32,767. Or, the value is not a positive integer value.

User Action. Correct the LINESIZE option and recompile the program.

NOCURREC No current record.

Informational. A DELETE or REWRITE statement was specified for a file opened with the UPDATE attribute, but the KEY option was not specified. These statements may omit the KEY option only if the "current record" contains a valid value.

User Action. Correct the statement in the source program and recompile.

NOFROM No FROM specified or buffer not allocated.

Informational. A REWRITE statement was specified without the FROM option. The REWRITE statement is valid without the FROM option only if a previous READ statement on the file specified the SET option to allocate a buffer and set a pointer to the record read.

User Action. Correct the previous READ statement for the file so that is specifies the SET option or correct the REWRITE statement so that is specifies the FROM option.

NOKEY No KEY or KEYFROM specified.

Informational. A keyed I/O statement must specify a KEY or KEYFROM option.

User Action. Correct the statement and recompile the program. If you are attempting sequential access to a file, verify that you have also specified SEQUENTIAL in the file's attribute list.

NOSHARE SHARED_READ or SHARED_WRITE conflicts with NO_SHARE.

Informational. The ENVIRONMENT options SHARED_READ and SHARED_WRITE permit read or write sharing on a file, but the NO_SHARE option prohibits all sharing.

User Action. Determine whether the file is to be accessed for sharing. If not, delete the option in error. If it is to be shared, delete the NO_SHARE option. Recompile the program.

NOTINDEXED Requested operation requires an INDEXED file.

Informational. A keyed I/O statement specifies an operation that is valid only for a file with indexed sequential file organization.

User Action. Determine from the information in the FILENAME message whether the operation was requested to the appropriate file. If the file is correctly specified but is not an indexed file, it may not have been properly created.

NOTINPUT Attempting to GET from an OUTPUT or UPDATE file.

Informational. A GET statement is not valid on a file that is opened with the OUTPUT or UPDATE attributes.

User Action. Correct the file's attribute list and recompile.

NOTKEYD Not a KEYED file.

Informational. A KEY or KEYFROM option was specified in a record I/O statement for a file that does not have the KEYED attribute.

User Action. Verify that the file is a keyed file, and if it is, correct the DECLARE or OPEN statement for the file so that it specifies the KEYED attribute.

NOTOUT Attempting to PUT to an INPUT or UPDATE file.

Informational. The PUT statement is not valid for files that are opened with the INPUT or UPDATE attribute.

User Action. Correct the file's attribute list and recompile.

PROMPTOBIG PROMPT option too long. Must be < 254 characters.

Informational. The string specified in the PROMPT option of the GET statements exceeds the maximum length of 253 characters.

User Action. Shorten the prompting string and recompile.

READOP PL/I compiler/run-time error. Please submit an SPR.

Informational. An error occurred in the execution of the PL/I compiler or a run-time module.

User Action. Gather as much information as possible about the circumstances under which the error occurred and submit an SPR report.

READOUT Attempting to READ from an OUTPUT file.

Informational. A file that is opened with the OUTPUT attribute cannot be accessed with a READ statement. If you are attempting to read a file that was just written, you must first close the file and reopen it with the INPUT attribute.

User Action. Correct the source program and recompile.

RECID File not open for RECORD_ID_ACCESS.

Informational. The RECORD_ID_TO and RECORD_ID_FROM options are valid only if the file's ENVIRONMENT option list specified RECORD_ID_ACCESS.

User Action. Correct the ENVIRONMENT option list and recompile the program.

RECIDKEY RECORD_ID_FROM conflicts with KEYED or KEYFROM.

Informational. A record I/O statement may not specify the KEY or KEYFROM option and the RECORD_ID_FROM option at the same time.

User Action. Correct the statement and recompile the program.

RECORD Record length does not match target length.

Informational. A fixed-length character string buffer is not the same length as a record being read by a READ statement.

User Action. Verify that the variable to which you are transferring data is the correct length for the records in the file. Correct the source program and recompile.

RECURSIO Illegal recursive I/O attempted.

Informational. An input or output operation was attempted to a file on which another I/O operation is currently being performed.

User Action. Correct the logic of the program and recompile.

STROVFL Stream item too big. Must be less than 1000 characters.

Informational. The run-time system cannot process a string longer than 1000 characters.

User Action. Correct the input or output field width and recompile the program. If necessary, use more than one stream I/O statement.

SUBRANGE Subscript range check error.

Error. The compiler detected a value that is beyond the range specified for a variable. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action. Correct the reference.

SUBSTR2 Operand two of a SUBSTR is out of range.

Error. The second operand in a reference to a SUBSTR built-in function or pseudovariable is beyond the range of the string. This message is issued only if the procedure containing this reference was compiled with the /CHECK qualifier.

User Action. Correct the reference.

SUBSTR3 Operand three of a SUBSTR is out of range.

Error. The third operand in a reference to a SUBSTR built-in function or pseudovariable is beyond the range of the string. This message is issued only if the procedure containing this reference was compiled with the /CHECK qualifier.

User Action. Correct the reference.

TITLE Invalid TITLE specified.

Informational. The size of the character-string expression specified in the TITLE option exceeds the maximum size of 128 bytes.

User Action. Select a smaller file title and correct the program.

VIRMEMDEAL PL/I compiler/run-time error. Please submit an SPR.

Informational. An error occurred in the execution of the PL/I compiler or a run-time module.

User Action. Gather as much information as possible about the circumstances under which the error occurred and submit an SPR report.

WRITEIN Attempting to WRITE to an INPUT file.

Informational. A file that is opened with the INPUT attribute cannot be accessed with a WRITE statement. If you are attempting to write a file that was just read, you must first close the file and reopen it either with the UDPATE attribute or with the OUTPUT attribute and ENVIRONMENT(APPEND).

User Action. Correct the source program and recompile.

Appendix C
Correspondence of PL/I and RMS

Table C-1 lists the VAX-11 PL/I ENVIRONMENT options and gives the VAX-11 RMS macro, field, or bit setting, as appropriate, that corresponds to each.

For detailed descriptions of the RMS fields, see the *VAX-11 Record Management Services (RMS) Reference Manual.*

Table C-1: RMS Fields for PL/I ENVIRONMENT Options

Options	RMS Macro/Field
APPEND	$RAB ROP=EOF $FAB FOP=CIF,- ^MXV,^NEF,^SUP
BATCH	$FAB FOP=SCF
BLOCK_BOUNDARY_FORMAT	$FAB RAT=BLK
BLOCK_IO	$FAB FAC=BIO
BLOCK_SIZE	$FAB BLS
BUCKET_SIZE	$FAB BKS
CARRIAGE_RETURN_FORMAT	$FAB RAT=CR
CONTIGUOUS	$FAB FOP=CTG
CONTIGUOUS_BEST_TRY	$FAB FOP=CBT
CREATION_DATE	$XABDAT CDT
CURRENT_POSITION	$FAB FOP=POS
DEFAULT_FILE_NAME	$FAB DNM
DEFERRED_WRITE	$FAB FOP=DFW
DELETE	$FAB FOP=DLT
EXPIRATION_DATE	$XABDAT EDT
EXTENSION_SIZE	$FAB DEQ

(Continued on next page)

Table C-1 (Cont.): RMS Fields for PL/I ENVIRONMENT Options

Options	RMS Macro/Field
TEMPORARY	$FAB FOP=TMP
TRUNCATE	$FAB FOP=TEF
WORLD_PROTECTION	$XABPRO PRO
WRITE_BEHIND	$RAB ROP=WBH
WRITE_CHECK	$FAB FOP=WCK

Appendix D
ASCII Character Set

ASCII Character Set

ASCII Decimal Number	Character	Meaning	ASCII Decimal Number	Character	Meaning
0	NUL	Null	40	(	Left parenthesis
1	SOH	Start of heading	41	)	Right parenthesis
2	STX	Start of text	42	*	Asterisk
3	ETX	End of text	43	+	Plus sign
4	EOT	End of transmission	44	,	Comma
5	ENQ	Enquiry	45	-	Minus sign or hyphen
6	ACK	Acknowledgement	46	.	Period or decimal point
7	BEL	Bell	47	/	Slash
8	BS	Backspace	48	0	Zero
9	HT	Horizontal tab	49	1	One
10	LF	Line feed	50	2	Two
11	VT	Vertical tab	51	3	Three
12	FF	Form feed	52	4	Four
13	CR	Carriage return	53	5	Five
14	SO	Shift out	54	6	Six
15	SI	Shift in	55	7	Seven
16	DLE	Data link escape	56	8	Eight
17	DC1	Device control 1	57	9	Nine
18	DC2	Device control 2	58	:	Colon
19	DC3	Device control 3	59	;	Semicolon
20	DC4	Device control 4	60	<	Left angle bracket
21	NAK	Negative acknowledgement	61	=	Equal sign
22	SYN	Synchronous idle	62	>	Right angle bracket
23	ETB	End of transmission block	63	?	Question mark
24	CAN	Cancel	64	@	At sign
25	EM	End of medium	65	A	Upper case A
26	SUB	Substitute	66	B	Upper case B
27	ESC	Escape	67	C	Upper case C
28	FS	File separator	68	D	Upper case D
29	GS	Group separator	69	E	Upper case E
30	RS	Record separator	70	F	Upper case F
31	US	Unit separator	71	G	Upper case G
32	SP	Space or blank	72	H	Upper case H
33	!	Exclamation mark	73	I	Upper case I
34	"	Quotation mark	74	J	Upper case J
35	#	Number sign	75	K	Upper case K
36	$	Dollar sign	76	L	Upper case L
37	%	Percent sign	77	M	Upper case M
38	&	Ampersand	78	N	Upper case N
39	'	Apostrophe	79	O	Upper case O

(Continued on next page)

Index

A

E

G

J

K

L

M

P

S

T

U

VAX-11 PL/I
User's Guide
AA-H951A-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_____

digital

PRINTED IN USA