PL/I

For OpenVMS Systems User Manual

Order Number: FIELD_TEST

September 1993

This manual provides an overview of the PL/I language and explains PL/I programming on the OpenVMS VAX and OpenVMS AXP operating systems. It describes the operation of the VAX PL/I and DEC PL/I compilers and the features of the operating systems that are important to the PL/I programmer.

Revision/Update Information: This revised manual supersedes the *PL/I User's Manual for VAX VMS*, Order Number AA–H951D–TE.

Operating System and Version: For VAX PL/I: OpenVMS Version 5.4 and higher versions

For DEC PL/I: OpenVMS AXP Version 1.5 and higher versions

Software Version: VAX PL/I Version 3.5
DEC PL/I Version 4.0

Digital Equipment Corporation
Maynard, Massachusetts

The postpaid Reader's Comments form at the end of this manual requests your critical evaluation to assist in preparing future documentation.

This document is available on CD–ROM.

Contents

4 The File System

5 Stream Input/Output

6 Record Input/Output

7 Options of the ENVIRONMENT Attribute

11 Using PL/I in the Common Language Environment

B Correspondence of PL/I and RMS

C Optional Programming Productivity Tools

D Rules for Conversion of Data

E The VAX Common Data Dictionary

Index

Examples

Figures

Tables

Preface

Manual Objectives

This manual describes how to use the PL/I compiler on the VMS and OpenVMS operating systems and contains detailed explanations of the extensions made to the standard PL/I language for PL/I for OpenVMS VAX and PL/I for OpenVMS AXP. To aid in program development, it includes information on some commands and utilities in the operating systems. It also includes information to assist in writing PL/I programs that use features of the file system and the operating systems.

Intended Audience

This manual is designed for programmers who have a working knowledge of PL/I and some familiarity with the VMS and OpenVMS operating systems and the DIGITAL Command Language (DCL).

Associated Documents

The *PL/I for OpenVMS Systems Reference Manual* contains a complete definition of the PL/I language, with detailed reference information on all standard PL/I language elements.

The *PL/I for OpenVMS Systems Installation Guide* gives instructions on how to install the PL/I compilers.

The OpenVMS documentation set gives complete information on the OpenVMS operating system.

On-Line Examples

All the full program examples in this manual, and some additional examples, are on line in SYS$COMMON:[SYSHLP.EXAMPLES.PLI]. (This device-directory specification may have been given the logical name PLI$EXAMPLES during installation of the PL/I compiler.)

Conventions

Return	A symbol with a key name indicates that you press a key on the terminal, for example, Return or ESC.
Ctrl/x	The symbol Ctrl/x indicates that you press the key "x" while holding down the key labeled Ctrl, for example, Ctrl/C.
italics	Italics indicate the introduction of a term.
$ **bold** Return	In interactive dialogues between the system and the user, user input is printed in boldface characters.
. . .	Vertical ellipses indicate that not all of the text of a program or program output is illustrated. Only relevant material is shown in the example.
option, . . .	Horizontal ellipses indicate that additional optional parameters, options, or values are allowed. When a comma precedes an ellipsis, it indicates that successive items must be separated by commas.
quotation mark apostrophe	The term quotation mark is used only to refer to the double quotation mark character ("). The term apostrophe is used to refer to the single quotation mark character (′).
[OPTIONS (option, . . .)]	Except in OpenVMS file specifications, brackets indicate that a syntactic element is optional and you need not specify it.
[LIST [EDIT	Brackets surrounding two or more stacked items indicate conflicting options, one of which *can* optionally be chosen.
{ EXTERNAL } { INTERNAL }	Braces surrounding two or more stacked items indicate conflicting options, one of which *must* be chosen.
FILE (file-reference)	An uppercase word or phrase indicates a keyword that must be entered as shown; a lowercase word or phrase indicates an item for which a variable value must be supplied. This convention applies to format (syntax) lines, not to code examples.

Terminological Assumptions

Information in this manual applies to the use of PL/I the OpenVMS VAX Operating System and the OpenVMS AXP Operating System unless otherwise indicated.

The terms *PL/I for OpenVMS VAX* and *VAX PL/I* are synonymous. Similarly, the terms *PL/I for OpenVMS AXP* and *DEC PL/I* are synonymous.

1
Overview

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP are implementations of a comprehensive and powerful programming language useful for systems programming, scientific computation, and commercial data handling and data organization. PL/I is a block-structured language that lends itself to the creation of efficient and maintainable structured programs. It has extensive string-handling capabilities. The PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compilers generate optimized, position-independent machine code.

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP are Digital's implementations, with extensions, of the American National Standard PL/I General-Purpose Subset, which is ANSI X3.74-1981. The General-Purpose Subset is a subset of full PL/I, which is ANSI X3.53-1976.

The General-Purpose Subset was designed for scientific, commercial, and systems programming on small and medium-size computer systems. The subset includes features of full PL/I that are easy to learn and use. It excludes features of full PL/I that are difficult to learn, prone to error, or not often used, and that greatly increase the complexity of run-time support required by the compiler.

Over and above its conformity to the General-Purpose Subset, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP offer many enhancements, which are extensions to that subset. Some of these extensions are features of full PL/I, and some are features for compatibility with other PL/I implementations in the industry. Other extensions integrate PL/I for OpenVMS VAX and PL/I for OpenVMS AXP into the VAX common language environment and into a complete programming environment. In general, the PL/I for OpenVMS VAX and and PL/I for OpenVMS AXP extensions are intended for programs that will execute exclusively under the control of the VMS operating system.

The PL/I for OpenVMS VAX and PL/I for OpenVMS AXP extensions provide for support of the OpenVMS Calling Standard, allowing PL/I procedures to call procedures written in other languages. The extensions provide access to the file capabilities of VAX Record Management Services, the OpenVMS file system; they provide improved condition handling; and they provide access to system services, which are operating-system procedures, and to run-time library procedures. Another notable extension is the preprocessor, which allows for conditional compilation of programs and for compile-time source transformation; it enables users to write procedures that will be executed at compile time.

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP can be used with a number of tools in a complete programming environment. For example, the VMS Debugger is a powerful tool for debugging programs. Use of the debugger is described in Chapter 3 of this manual. Other tools are optional products, including VAX DATATRIEVE, the VAX Common Data Dictionary, and the VAX Software Engineering Tools (VAXset).

VAXset includes several integrated components, including the VAX Code Management System (CMS), the VAX DEC/Test Manager, the VAX Source Code Analyzer (SCA), the VAX Program Design Facility (PDF), and the VAX Language-Sensitive Editor (LSE). Notably, the VAX Language-Sensitive Editor can be a valuable aid when you write and compile a program, as it includes built-in intelligence about the source format and syntax of PL/I programs. Contact your system manager to find out which of these and other optional products are installed on your system.

Digital's implementations of the PL/I language are fully described in the *PL/I for OpenVMS Systems Reference Manual*. That manual contains further information on the General-Purpose Subset, the OpenVMS VAX and OpenVMS AXP extensions, and the differences between Digital's implementations and other implementations of the PL/I language.

2

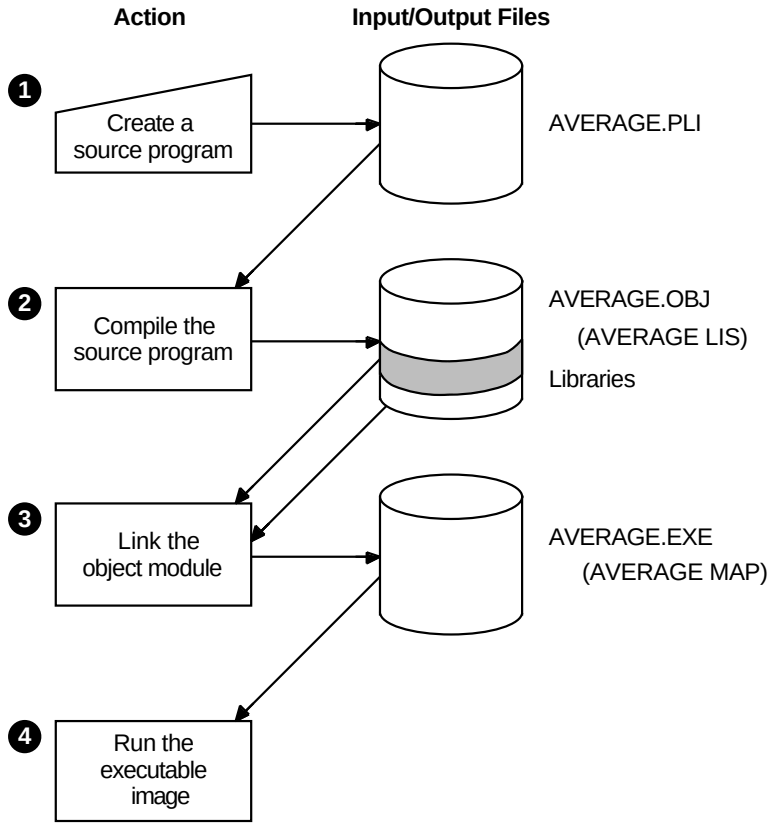
Developing PL/I Programs at the DCL Command Level

This chapter describes how to create, compile, link, and run an OpenVMS VAX or PL/I for OpenVMS AXP program using DCL commands.

2.1 DCL Commands for Program Development

This section describes the DCL commands that are used to create, compile, link, and run a PL/I program on an OpenVMS system. These commands are shown in Figure 2–1. The following sections describe these commands in more detail.

Figure 2–1 DCL Commands for Developing Programs



The following example shows the commands to perform the actions shown in Figure 2–1:

```
$ EDIT/EDT AVERAGE.PLI 1
$ PLI AVERAGE          2
$ LINK AVERAGE         3
$ RUN AVERAGE          4
```

To create a PL/I source program at the DCL level, you must invoke a text editor. In the previous example, the VAX EDT editor is invoked to create the source program AVERAGE.PLI. You can, however, use another editor, such as the VAX Language-Sensitive Editor (LSE) in VAXset. PLI is used as the file type to indicate that you are creating a PL/I source program. PLI is the conventional file type for all PL/I source programs.

When you compile your program with the PLI command, you do not have to specify the file type; the PL/I for OpenVMS VAX compiler searches for PLI by default.

If your source program compiles successfully, the PL/I for OpenVMS VAX compiler creates an object file with the file type OBJ.

However, if the PL/I for OpenVMS VAX compiler detects errors in your source program, the system displays each error on your screen and then displays the DCL prompt. You can then reinvoke your text editor to correct the errors.

You can include command qualifiers with the PLI command. Command qualifiers cause the PL/I for OpenVMS VAX compiler to perform additional actions. In the following example, the /LIST qualifier causes the PL/I for OpenVMS VAX compiler to produce a listing file.

```
$ PLI/LIST AVERAGE
```

For a complete list and explanation of all of the command qualifiers available with the PLI command, see Section 2.3.2.

Once your program has compiled successfully, you invoke the VMS Linker to create an executable image file. The VMS Linker uses the object file produced by PL/I for OpenVMS VAX as input to produce an executable image file as output.

You can specify command qualifiers with the DCL command LINK. For a complete list and explanation of all the command qualifiers available with the LINK command, see Section 2.4.2.

Once the executable image file has been created, you can run your program with the DCL command RUN.

2.2 Creating a PL/I Program

To create and modify a PL/I program, you must invoke a text editor. The OpenVMS system supports the following text editors: VAX EDT (EDT) and the Language-Sensitive Editor (LSE). The following sections describe briefly how to use both EDT and LSE. LSE is a layered extension of the older VAXTPU, which is also described briefly.

2.2.1 Using VAX EDT

EDT is an interactive general-purpose text editor that offers three editing modes: keypad, nokeypad, and line. With keypad mode, you issue commands by using the numeric keypad that appears to the right of your main keyboard. With nokeypad mode, you issue commands on a command line, which EDT processes when you press the Return key. Line mode focuses on the line as the unit of text. With line mode, you issue commands at the line mode asterisk prompt (*).

Keypad mode and nokeypad mode continually display the contents of the file on your screen. When you begin your editing session, editing in line mode is the default. Unlike keypad and nokeypad mode, line mode displays only one line of text on your screen.

The following command line invokes the EDT editor and creates the file PROG_1.PLI.

```
$ EDIT/EDT PROG_1.PLI
```

To change from line mode to keypad mode, type the CHANGE command at the asterisk prompt. To return to line mode from keypad mode, press Ctrl/z. To change from line mode to nokeypad mode, type the SET NOKEYPAD command and then type the CHANGE command.

When you invoke EDT to create a file, a journal file is created automatically. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, type the EDIT/RECOVER command.

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

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

2.2.2 Using VAX LSE

The VAX Language-Sensitive Editor (LSE), a component of VAX Software Engineering Tools (VAXset), is a multilanguage, programmable editor designed specifically to help develop and maintain source code. LSE is layered on top of the VAX Text Processing Utility (VAXTPU), and is available with the EVE and EDT interfaces. LSE provides language-specific templates for each language it supports, including PL/I; these templates help the programmer build syntactically correct programs efficiently.

LSE provides the following features:

- Language-specific source code templates for fast entry of source code
- Compilation, review, and correction of compile-time errors within a single edit session
- Interactive edit capabilities while debugging
- Ability to customize editing environments
- Integrated access to Source Code Analyzer (SCA) cross-referencing features
- Ability to access SCA or diagnostics file information

- Support for a package facility to define your own subroutine call templates, with tokens and placeholders available for use by multiple LSE environments
- Source code templates for calls to VMS system services
- Support for user-written diagnostic files, enhancing support for nonsupported and user-modified compilers
- Besides PL/I, LSE supports the following VAX products:

  VAX Ada
  VAX BASIC
  VAX BLISS-32
  VAX C
  VAX CDD/Plus
  VAX COBOL
  VAX DATATRIEVE
  VAX DIBOL
  VAX DOCUMENT
  VAX FORTRAN
  VAX MACRO
  VAX Pascal
  VAXELN Pascal
  VAX SCAN

2.2.3 Using VAXTPU

The VAX Text Processing Utility (VAXTPU) is a high-performance, programmable utility. VAXTPU provides two editing interfaces: the Extensible VAX Editor (EVE) and the VAXTPU EDT Keypad Emulator. You can also create your own interfaces.

Like EDT, VAXTPU provides you with an online HELP facility that you can access during your editing session. When you invoke VAXTPU to create a file, a journal file is created automatically. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, include the /RECOVER qualifier on the command line that invokes the editor.

Unlike EDT, VAXTPU provides multiple windows. This feature allows you to view two files on your screen at the same time.

The following sections describe how to use the EVE interface and the EDT Keypad Emulator interface.

2.2.3.1 The EVE Interface

EVE is an interactive text editor that allows you to execute common editing functions using the EVE keypad or to execute more advanced functions by typing commands on the EVE command line. The following command line invokes the EVE editor and creates the file PROG_1.PLI:

```
$ EDIT/TPU PROG_1.PLI
```

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

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

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

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

2.2.3.2 The EDT Keypad Emulator Interface

The EDT Keypad Emulator interface provides all of the keypad functions associated with EDT and uses the same keys to perform each function. It also provides a subset of the line editing commands of the EDT editor; however, it does not provide nokeypad editing. The following command line invokes the EDT Keypad Emulator:

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

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

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

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

```
$ EDTEM PROG_1.PLI
```

For more detailed information on how to use the EDT Keypad Emulator, see the *Guide to VMS Text Processing*.

2.3 Compiling a PL/I Program

The PL/I for OpenVMS VAX compiler performs the following functions:

- Detects errors in your source program
- Displays each error on your screen or writes the errors to a file
- Generates machine language instructions from the source statements
- Groups these language instructions into an object module for the VMS Linker

The following sections discuss the PLI command and its qualifiers.

2.3.1 PLI Command

To invoke the PL/I for OpenVMS VAX compiler, use the PLI command. The PLI command has the following format:

```
PLI[/qualifier...][ file-spec [/qualifier...]],...
```

/qualifier
Specifies an action to be performed by the compiler on all files or specific files listed. When a qualifier appears directly after the PLI command, it affects all files listed. However, when a qualifier appears after a file specification, it affects only the file that immediately precedes it. When files are concatenated, however, these rules do not apply.

file-spec
Specifies an input source file that contains the program or module to be compiled. You are not required to specify a file type; the PL/I for OpenVMS VAX compiler adopts the default file type, PLI.

You can include more than one file specification on the same command line by separating the file specifications with either a comma (,) or a plus sign (+). If you separate the file specifications with commas, you can control which source files are affected by each qualifier. In the following example, the PL/I for

OpenVMS VAX compiler creates an object file for each source file but creates only a listing file for the source files PROG_1 and PROG_3.

```
$ PLI /LIST PROG_1, PROG_2/NOLIST, PROG_3
```

If you separate the file specifications with plus signs, the PL/I for OpenVMS VAX compiler concatenates each of the specified source files and creates one object file and one listing file. In the following example, only one object file is created, PROG_1.OBJ, and only one listing file is created, PROG_1.LIS. Both of these files are named after the first source file in the list, but contain all three modules.

```
$ PLI PROG_1 + PROG_2/LIST + PROG_3
```

Note that any qualifiers specified for a single file within a list of files separated with plus signs affect all the files in the list.

You can specify a library file in a PLI command. To do this, you must precede the specification with a plus sign and use the /LIBRARY qualifier. For example:

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

This PLI command compiles the source program APPLIC.PLI and uses the library DATAB.TLB to locate any INCLUDE files that are referenced in the following format:

```
%INCLUDE 'text-module-name';
```

The module name must not be enclosed in apostrophes.

When you specify more than one library, PL/I searches the libraries in the order specified each time it processes a %INCLUDE statement that specifies a text module name. For example:

```
$ PLI APPLIC+DATAB/LIBRARY -
$_+NAMES/LIBRARY+GLOBALSYMS/LIBRARY
```

When PL/I processes a %INCLUDE statement in the source file APPLIC.PLI, it searches for modules referenced in the libraries DATAB.TLB, NAMES.TLB, and GLOBALSYMS.TLB, in that order.

On a command that requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

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

In this example, PL/I compiles METRIC.PLI and APPLIC.PLI separately and uses the library DATAB.TLB for each compilation.

The order of appearance of the library file specification within a concatenated list of files is irrelevant. For example, the following are equivalent:

```
$ PLI ALPHA+MYLIB/LIBRARY+BETA
$ PLI ALPHA+BETA+MYLIB/LIBRARY
```

You can define one of your private INCLUDE file libraries as a default library for the PL/I compiler to search. The compiler searches the default library after it searches libraries specified on the PLI command.

To define a default library, define an equivalence for the logical name PLI$LIBRARY, as in the following example:

```
$ DEFINE PLI$LIBRARY DATAB
```

While this assignment is in effect, the compiler automatically searches the library DATAB.TLB for any INCLUDE modules that it cannot locate in libraries explicitly specified on the PLI command.

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.

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$LIBRARY:PLI$STARLET.TLB
```

SYS$LIBRARY is normally defined by the system manager to identify the device and directory containing system libraries. PLI$STARLET.TLB is a library of INCLUDE modules supplied by OpenVMS VAX and PL/I for OpenVMS AXP. It contains declarations for the entry points for OpenVMS system services, local symbol definitions required for use with them, and variables to test their return status values. For more information on libraries, see the *VMS Librarian Utility Manual*.

2.3.2 PLI Command Qualifiers

The qualifiers available with the PLI command are listed and explained in this section.

Command Qualifier	Default
/[NO]ALIGN[=option][1]	/NOALIGN
/[NO]ANALYSIS_DATA[=file-spec][3]	/NOANALYSIS_DATA
/[NO]CHECK	/NOCHECK
/CHECK[=option]	/CHECK=ALL
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/DATA[=option]	/DATA=NATIVE
/[NO]DEBUG[1]	/NODEBUG
/DEBUG[=option][1]	/DEBUG=ALL
/[NO]DESIGN[=option,...][3]	/NODESIGN
/[NO]DIAGNOSTICS	/NODIAGNOSTICS
/[NO]ERROR_LIMIT	/NOERROR_LIMIT
/ERROR_LIMIT[=n]	/ERROR_LIMIT=100
/FIXED_BINARY[=p]	/FIXED_BINARY=31
/FLOAT[=option][2]	/FLOAT=D_FLOAT
/[NO]G_FLOAT	/NOG_FLOAT
/GRANULARITY[=option][2]	/GRANULARITY=QUADWORD
/[NO]LIST[=file-spec]	/NOLIST *(interactive default)* /LIST *(batch default)*
/MACHINE_CODE[=option]	/MACHINE_CODE=INTERSPERSED *(OpenVMS VAX)* AFTER *(OpenVMS AXP)*
/[NO]OBJECT[=file-spec][1]	/OBJECT
/[NO]OPTIMIZE[=option,...]	/OPTIMIZE=ALL *(OpenVMS VAX)* (LEVEL=4,UNROLL=0) *(OpenVMS AXP)*
/SHOW[=option,...]	/SHOW=NOINCLUDE NODICTIONARY NOMAP SOURCE NOTRACE TERMINAL HEADER *(OpenVMS AXP only)* NOEXPANSION NOSTATISTICS
/VARIANT[= ["]alphanum_string["]	/VARIANT=""
/[NO]WARNINGS	/WARNINGS[=(option,...)]

Certain pairs of command qualifiers, as follows, are mutually exclusive and should not be used together in the same command line.

- /FLOAT and /G_FLOAT
- /DATA and /ALIGN
- /DATA and /FIXED_BINARY

[1] Options differ for OpenVMS VAX and OpenVMS AXP.
[2] Command qualifier is valid for OpenVMS AXP only.
[3] Command qualifier is valid for OpenVMS VAX only.

Command qualifiers request processing options of the compiler. You can specify qualifiers to the PLI command after the command name or an individual file specification. When a qualifier is specified after 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 after a file specification in a list of files separated by commas, its action is applied only to the compilation of that file.

/ALIGN
/NOALIGN (default)
Controls alignment of data within structures and aligned bit strings. If you specify /ALIGN on a VAX, data is aligned on the natural byte boundary of the specified data type, as shown in Table 2–1. If you specify /NOALIGN, the default, data is aligned on the next available byte boundary.

Two options are provided for PL/I for OpenVMS AXP only: *packed* and *natural*.

If you specify /ALIGN=natural, data is aligned on the natural byte boundary of the specified data type, as shown in Table 2–1. If you specify /ALIGN=packed, data is aligned on the next available byte boundary.

Table 2–1 Natural Data Alignment

Data Type	Data Size	Alignment
fixed bin (p)	p <= 7	byte
fixed bin (p)	7 < p <= 15	word
fixed bin (p)	p > 15	longword
fixed dec (p,q)		word
float bin (p)	p <= 24	longword
float bin (p)	24 < p <= 53	quadword
float bin (p)	p > 53	octaword
float dec (p)	p <= 7	longword
float dec (p)	7 < p <= 15	quadword
float dec (p)	p > 15	octaword
char	N/A	byte
char aligned	N/A	byte
char varying	N/A	word
bit	N/A	bit
bit aligned	N/A	longword
pointer	c	longword
area		longword
label	c	quadword
entry	c	quadword
file	c	longword

(continued on next page)

Table 2–1 (Cont.) Natural Data Alignment

Data Type	Data Size	Alignment
structure	varies	max of members
picture	N/A	byte
offset	c	longword

/ANALYSIS_DATA[=file-spec]
/NOANALYSIS_DATA (default)
Controls whether the compiler generates a file of source code analysis information. The default file name is the file name of the primary source file; the default file type is ANA.

/CHECK
/NOCHECK (default)
Controls the checking of array subscripts and of positional references in arguments to the SUBSTR built-in function. If you specify /CHECK, the compiler checks for the following conditions:

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

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

Specifying /CHECK is equivalent to specifying /CHECK=ALL or the attribute /CHECK=BOUNDS. Likewise, /NOCHECK is the equivalent of specifying /CHECK=NONE or the attribute /CHECK=NOBOUNDS.

/CROSS_REFERENCE
/NOCROSS_REFERENCE (default)
Specifies whether the compiler is to generate, in the listing file, alphabetical cross-references for variable names. If you specify /CROSS_REFERENCE, the compiler lists all variable names, including all members of structures as separate entities in an alphabetical cross-reference listing. The cross-reference entry for each structure member also lists the name of the structure that contains the member. The listing contains the line numbers of the lines on which all variables are referenced.

Note that you must specify /SHOW=MAP with /CROSS_REFERENCE. The full specification is as follows:

```
$ PLI/LIST/SHOW=MAP/CROSS_REFERENCE file.PLI
```

By default, the compiler does not include cross-references in the listing.

/DATA[=option] (OpenVMS AXP only)
Specifies the integer size and whether alignment is enabled. You can select the following options:

NATIVE	Sets the integer size to 32 and enables /ALIGNMENT; equivalent to ALPHA_AXP32 or MIPS on an OpenVMS AXP system.
VAX	Sets the integer size to 32 and enables /NOALIGNMENT
ALPHA_AXP32	Sets the integer size to 32 and enables /ALIGNMENT

/DEBUG[=option]
/NODEBUG (default)
Requests that information be included in the object module for use with the OpenVMS Debugger.

It is strongly recommended that you use the /NOOPTIMIZE qualifier when you use the /DEBUG qualifier. Optimization can cause confusing results during a debugging session.

When /DEBUG is specified, the compiler generates small routines that are used only by the debugger. When the program is linked with /NODEBUG, these routines will still be included in the program but will not be used. For this reason it is recommended that final versions of programs be recompiled with /NODEBUG (which is the default).

You can select the following options:

ALL	Includes symbol table records and traceback records. /DEBUG is equivalent to /DEBUG=ALL.
SYMBOLS	Includes the symbol definitions for all identifiers in the compilation. This is the default for symbols if the /DEBUG qualifier is used.
NOSYMBOLS	Does not include symbol definitions. Without symbol definitions, traceback is done according to virtual address.
TRACEBACK	Includes only traceback records. This is the default if the /DEBUG qualifier is not present on the command.
NOTRACEBACK	Does not include traceback records.
INLINE (OpenVMS VAX only)	Generates debug information to cause a STEP command to STEP /INTO an inlined fuction call.
NOINLINE (OpenVMS VAX only)	Generates debug information to cause a STEP command to STEP /OVER an inlined fuction call.
NONE	Does not include any debugging information. Use this option to exclude all debug information from thoroughly debugged program modules. /NODEBUG is equivalent to /DEBUG=NONE.

/DESIGN[=(option, . . .)](OpenVMS VAX only)
/NODESIGN (default)
Controls whether the compiler processes the source file as a detailed design, in conjunction with the VAXset or DECset Program Design Facility (PDF). The /DESIGN qualifier requires that the /ANALYSIS_DATA also be specified.

If you specify the /DESIGN qualifier, the compiler modifies its parsing and semantics according to the optional keywords you supply. The design information is added to the SCA Analysis Data File (.ANA).

You can select from the following options:

[NO]COMMENTS	COMMENTS, the default, directs the compiler to search inside comment fields for program design information. Specifying the option NOCOMMENTS omits this searching.
[NO]PLACEHOLDERS	PLACEHOLDERS, the default, directs the compiler to treat Language-Sensitive Editor (LSE) placeholders in proper contexts within the program as valid syntax. Specifying NOPLACEHOLDERS causes the compiler to treat all LSE placeholders as invalid syntax.

The options are mutually independent. Specifying /DESIGN without naming an option is equivalent to /DESIGN=(COMMENTS,PLACEHOLDERS). You can exclude both options by specifying /NODESIGN or by omitting the qualifier. To select one option and exclude the other, you must specify /DESIGN=NOCOMMENTS or /DESIGN=NOPLACEHOLDERS.

/DIAGNOSTICS[=file-spec]
/NODIAGNOSTICS (default)
Controls whether the compiler produces a diagnostics file, which contains compiler messages and diagnostic information.

The file type DIA is the default file type for a diagnostics file. The diagnostics file is used by Digital layered products such as the Language-Sensitive Editor.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT
Permits you to specify the number of errors acceptable during program compilation. For example, if you specify /ERROR_LIMIT=5, then compilation terminates with the sixth error. By default, compilation terminates when the number of errors exceeds 100, but /NOERROR_LIMIT raises the default number to 1000. The maximum number of error messages permitted by the system is 32,767.

All error messages (excluding warning messages) are counted toward the error limit. A fatal error message immediately terminates the compilation.

/FIXED_BINARY[=p]
This qualifier sets the default precision of fixed binary variables. You can specify either 15 or 31 as the value of p. If you omit this qualifier or specify it without a precision value, the default precision of fixed binary variables is 31. Normally, /FIXED_BINARY=15 is specified when compiling for a system whose memory words are 16 bits long.

/FLOAT[=option] (OpenVMS AXP only)
For AXP computers, specifies the default representation of floating-point variables. Precision depends upon whether the variable is binary or decimal. See *PL/I for OpenVMS Systems Reference Manual* for more information.

The qualifiers /FLOAT and /G_FLOAT are mutually exclusive and should not be used in the same command line.

/G_FLOAT
/NOG_FLOAT (default)
For OpenVMS VAX and OpenVMS AXP systems equipped with the appropriate hardware option, specifies the default representation of floating-point variables, depending upon whether the variable is binary or decimal.

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 *PL/I for OpenVMS Systems Reference Manual*.

/GRANULARITY[=option] (OpenVMS AXP only)
Specifies the smallest unit of data that can be cached in a register. The following options are available:

- byte
- longword
- quadword

For example, quadword granularity would mean that the AXP system would cache quadword values in registers. The default granularity is quadword.

/LIST[=file-spec] (batch default)
/NOLIST (interactive default)
Controls whether a listing file is produced.

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.

/MACHINE_CODE[=option]
/NOMACHINE_CODE (default)
Controls whether the listing file produced by the compiler includes a listing of the machine code generated during the compilation.

You can select the following options:

AFTER	Puts machine code after the source code (the default if /MACHINE_CODE is specified on an AXP machine)
BEFORE	Puts machine code before the source code
INTERSPERSED (OpenVMS VAX only)	Intersperses source and machine code (the default if /MACHINE_CODE is specified on a VAX machine)

No machine code is generated if /NOOBJECT is specified; thus, if /NOOBJECT /MACHINE_CODE are specified together, no machine code listing is generated.

/OBJECT[=file-spec] (default)
/NOOBJECT
Controls whether the compiler produces object modules. By default, the compiler produces an object module with the same file name as the source file and a file type of OBJ.

Specify /NOOBJECT when you want to compile a program only to obtain a listing or when you want the compiler only to check the source program for errors and display diagnostic messages. The compiler can compile code more rapidly if it does not need to create an object module.

/OPTIMIZE[=(option,...)]
/NOOPTIMIZE (OpenVMS VAX only)
Controls the optimization performed by the compiler. On an OpenVMS VAX system, you can select one or more of the options listed in Table 2–2. By default, all possible optimizations are performed.

If you specify /OPTIMIZE with any options, the other options are not affected. For example, /OPTIMIZE=NOPEEPHOLE disables the PEEPHOLE option, but leaves all other options enabled.

Table 2–2 Compiler Optimization Options for VAX Systems

Option	Description
ALL	Performs all optimizations. This is equivalent to /OPTIMIZE and is the default.
[NO]COMMON_SUBEXPRESSIONS	Eliminates or does not eliminate common subexpressions.
[NO]DISJOINT	Places or does not place local variables in multiple registers.
[NO]INLINE	Provides or does not provide automatic inline expansion of procedures to provide optimized code.
[NO]INVARIANT	Removes or does not remove invariant expressions from loops.
[NO]LOCALS_IN_REGISTERS	Places or does not place local variables in registers.
[NO]PEEPHOLE	Performs or does not perform pattern replacement on the generated machine code.
[NO]RESULT_INCORPORATION	Collapses or does not collapse binary arithmetic operations into 3-operand instructions.

Note that if you specify NOCOMMON_SUBEXPRESSION with the /OPTIMIZE qualifier to the PLI command, NORESULT_INCORPORATION is implied.

/OPTIMIZE[=(option=n,...)]
/NOOPTIMIZE (OpenVMS AXP)
Controls the optimization performed by the compiler. On an OpenVMS AXP system, you can specify a value for the options listed in Table 2–3.

Table 2–3 Compiler Optimization Options for AXP Systems

Option	Description
LEVEL=n	Specifies the level of optimization based on a value from 0 to 5. Zero (0) specifies no optimization and is equivalent to /NOOPTIMIZE. The default value for LEVEL is 4.

(continued on next page)

Table 2–3 (Cont.) Compiler Optimization Options for AXP Systems

Option	Description
UNROLL=n	Specifies the number of times to unroll loops, based on a value from 0 to 16. Causes GEM to use its default /UNROLL value, which is 4. If the /UNROLL option is not specified, a default unroll amount of 4 is used. LEVEL must be specified as 3 or greater in order to enable loop unrolling.
INLINE=NONE MANUAL AUTOMATIC ALL	Provides or does not provide automatic inline expansion of procedures to provide optimized code.

Each subsequent level of optimization includes the optimizations from all lower levels. The levels of optimization are as follows:

- 0 — No optimizations performed. The only code transformations performed are block reordering, and intermediate language and final peepholes.
- 1 — Optimizations performed that do not seriously affect readability or debugging. This includes common subexpression and register history, and user-directed inlining.
- 2 — Optimizations performed that do not increase code size. This includes code motion, strength reduction, induction variables, test replacement, split lifetime analysis, and intermediate language and final scheduling.
- 3 — All optimizations performed except automatic inlining, including while-repeat transformations, loop unrolling, and final code replication.
- 4 — Full optimization, including automatic inlining. This is the default optimization level.
- 5 — Full optimization, as in level 4, but with more agressive inlining. This allows you to indicate that speed is more important than size.

/SHOW[=(option,...)]

Sets or cancels specific compilation listing options. You can select or cancel any of the options listed in Table 2–4.

Table 2–4 Compiler Listing Options

Option	Function
ALL	Includes the contents of all files and modules in the program listing.
NONE	Does not include the contents of any of the files and modules in the program listing.
[NO]INCLUDE	Includes or does not include the contents of INCLUDE files and modules in the program listing.
[NO]DICTIONARY	Includes or does not include the contents of Common Data Dictionary record modules in the program listing.

(continued on next page)

Table 2–4 (Cont.) Compiler Listing Options

Option	Function
[NO]MAP	Includes or does not include 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]SOURCE	Includes or does not include the source program statements in the program listing.
[NO]TERMINAL	Displays or does not display compilation messages to SYS$OUTPUT at compile time.
[NO]STATISTICS	Includes or does not include performance statistics in the program listing.
[NO]HEADER	Includes or does not include file headers in the program listing.
[NO]TRACE	Includes or does not include each step of preprocessor replacement and rescanning.
[NO]EXPANSION	Includes or does not include the final replacement values for preprocessor variables.

The following options are enabled by default:

NOINCLUDE
NOMAP
NODICTIONARY
SOURCE
TERMINAL
HEADER
NOSTATISTICS
NOTRACE
NOEXPANSION

The /SHOW qualifier must be used in combination with the /LIST qualifier to be effective. The /LIST qualifier specifies that a source listing is to be made, and the /SHOW qualifier gives you control over which portions of the source listing you want to see.

You can also control the content of the source listing by using preprocessor statements to suppress preprocessor portions in the program text. For example, if you previously specified /SHOW=INCLUDE, you can suspend included files from the listing with the %NOLIST_INCLUDE statement in your program.

By default, the /SHOW qualifier yields a listing with two items (P and ∗) noted

PL/I for OpenVMS VAX

in the column to the right of the line numbers.

If you specify /LIST/SHOW=ALL, the compiler includes the full complement of character notations

PL/I for OpenVMS AXP

at the beginning of the line, followed by the line numbers.

When you specify any option with the /SHOW qualifier, the settings for other options are not changed.

Table 2–5 summarizes the character notations that can appear in the listing.

Table 2–5 Character Notations That Can Appear in a Listing

Character	Qualifiers	Meaning
| (vertical bar)	/LIST	Indicates a line that contains a comment only.
* (asterisk)	/LIST	Indicates program text that was not used at compile time.
" (quotation mark)	/LIST/SHOW=EXPANSION	Indicates a continuation of a previous line wrapped at the right margin, to show the complete final replacement value of a preprocessor expansion.
+ (plus sign)	/LIST/SHOW=TRACE	Indicates flow of preprocessor procedure evaluation and out-of-sequence source processing resulting from %GOTO.
D	/LIST/SHOW=DICTIONARY	Indicates CDD text included by a %DICTIONARY statement.
E	/LIST/SHOW=EXPANSION	Indicates the final replacement value of a preprocessor variable or procedure.
I	/LIST/SHOW=INCLUDE	Indicates text included by a %INCLUDE statement.
P	/LIST	Indicates lines contained within a preprocessor procedure.
T	/LIST/SHOW=TRACE	Indicates each step of preprocessor replacement and rescanning.

/VARIANT
/VARIANT="" (default)

Permits specification of compilation variants. The value specified for /VARIANT is available at compile time with the VARIANT preprocessor built-in function.

If /VARIANT is not specified, or if /VARIANT is specified without a value, /VARIANT ="" is assumed.

/WARNINGS (default)
/WARNINGS=(option list)
/NOWARNINGS

Controls whether the compiler prints diagnostic warning and informational messages.

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

The /WARNINGS qualifier has two options:

- NOINFORMATIONALS causes the compiler to suppress informational messages.
- NOWARNINGS causes the compiler to suppress warning messages.

Note that the informational message SUMMARY cannot be suppressed with /NOWARNINGS or /WARNINGS=NOINFORMATIONALS.

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.

2.3.3 PL/I Preprocessor

The PL/I preprocessor permits you to alter a source program at compile time. Preprocessor statements can be mixed with nonpreprocessor statements in the source program, but preprocessor statements are executed only at compile time. The resulting source program is then used for further compilation.

The preprocessor performs two types of preprocessing:

- It interprets preprocessor statements and evaluates preprocessor expressions.
- It replaces the values of preprocessor variables and procedures.

Preprocessor statements allow you to include text from alternative sources (INCLUDE libraries and the VAX Common Data Dictionary), control the course of compilation (%DO, %GOTO, %PROCEDURE, and %IF), issue user-generated diagnostic messages, and selectively control listings and formats. The preprocessor statements are described in full in the *PL/I for OpenVMS Systems Reference Manual*.

2.3.3.1 Preprocessor Compilation Control

At compile time, preprocessor variables, procedures, and variable expressions are evaluated in the order in which they appear in the source text, and the new values are substituted in the source program in the same order. Thus, the course of compilation becomes conditional, and the resulting executable program may exhibit a variety of unique features. Note that preprocessor variables and procedures must be declared and activated before replacement occurs.

For example:

```
%DECLARE HOUR FIXED;
%HOUR = SUBSTR(TIME(),1,2);

%IF HOUR > 7 & HOUR < 18
%THEN
    %FATAL 'Please compile this outside of prime time';
%DECLARE T CHARACTER;
%ACTIVATE T NORESCAN;
%T = '''Compiled on '||DATE()||'''';
DECLARE INIT_MESSAGE CHARACTER(40) VARYING INITIAL(T);
```

```
    %IF VARIANT() = '' | VARIANT() = 'NORMAL'
    %THEN
        %INFORM 'NORMAL';
    %ELSE
        %IF VARIANT() = 'SPECIAL';
        %THEN
            %INFORM 'SPECIAL';
        %ELSE
            %IF VARIANT() = 'NONE';
            %THEN %;
            %ELSE
                %DO;
                %T = '''unknown variant''';
                %WARN T;
                INIT_MESSAGE = INIT_MESSAGE||' with '||T;
                %END;
    %END;
PUT LIST (INIT_MESSAGE);
```

This example illustrates several aspects of the preprocessor. First, this program must be compiled outside of prime time. Second, depending upon the value of VARIANT, the program is compiled with a different variant.

Notice the number of single quote marks around the string constant assigned to T. Single quotes are sufficient if the value of T is used only in a preprocessor user-generated diagnostic message. That is, the value of T is concatenated with nonpreprocessor text and assigned to INIT_MESSAGE because during preprocessing, single quotes are stripped from string constants. To ensure that the run-time program also has quotes around the string, additional quotes are needed.

2.3.3.2 Preprocessor Procedures

The %PROCEDURE statement defines the beginning of a preprocessor procedure block and specifies the parameters, if any, of the procedure. A preprocessor procedure executes only at compile time. Invocation is similar to a function reference and occurs in two ways:

- Preprocessor statements can invoke preprocessor procedures. In addition, preprocessor statements from within preprocessor procedures can invoke other preprocessor procedures.
- Statements from the source program can invoke preprocessor procedures.

A preprocessor procedure is invoked by the appearance of its entry name and list of arguments. If the reference occurs in a nonpreprocessor statement, the entry name must be active before the preprocessor procedure is invoked. If the entry name is activated with the RESCAN option, the value of the preprocessor procedure is rescanned for further possible preprocessor variable replacement and procedure invocation. Preprocessor procedures can be invoked recursively.

Since the preprocessor procedure is always invoked as a function, the %PROCEDURE statement must also specify (via the RETURNS option) the data type attributes of the value that is returned to the point of invocation.

The return value replaces the preprocessor procedure reference in the invoking source code. Preprocessor procedures cannot return values via their parameter list. The return value must be capable of being converted to one of the data types CHARACTER, FIXED, or BIT. The maximum precision of the value returned by the %RETURNS statement is BIT(31), CHARACTER(32500), and FIXED(10).

Preprocessor procedures can have one of two distinctly different types of argument lists: positional or keyword. Positional argument lists (ending with a right parenthesis) use parameters sequentially, as in a parenthesized list. Positional argument lists can be used in any preprocessor procedure. Keyword argument lists (ending with a semicolon) use parameters in any order, as long as each keyword matches the name of a parameter. Therefore, the order in which you list them does not affect the correct matching of parameters and arguments. Keyword argument lists can only be used when the preprocessor procedure contains the STATEMENT option and is invoked from a nonpreprocessor statement.

A keyword argument list ends with a semicolon rather than the right parenthesis. In this way, the STATEMENT option permits you to use a preprocessor procedure as if it were a statement. Consequently, preprocessor procedures using the STATEMENT option permit you to extend the PL/I language by simulating features that may not otherwise be available.

Preprocessor Statements

All preprocessor statements are preceded by a percent sign (%) and are terminated by a semicolon (;). All text that appears within these delimiters is considered part of the preprocessor statement and is executed at compile time. For example:

```
%DECLARE HOUR FIXED;          /* declaration of a preprocessor
                                 single variable */

%DECLARE (A,B) CHARACTER;     /* a factored preprocessor
                                 declaration     */

%HOUR = SUBSTR(TIME(),1,2);   /* preprocessor assignment
                                 statement using two built-in
                                 functions       */
```

Notice that a percent sign (%) is required only at the beginning of the statement. Preprocessor built-in functions are contained within preprocessor statements and consequently do not require a percent sign. However, when you include Common Data Dictionary record definitions, you may need to include the usual PL/I punctuation.

Labels are permitted on preprocessor statements and, like other PL/I labels, are used as the targets of program control statements. A preprocessor label must be an unsubscripted label constant and must be preceded by a percent sign. As with other preprocessor statements, the percent sign alerts the compiler that until the line is terminated with a semicolon, all subsequent text is preprocessor text. Therefore, no other percent signs are required on that line.

Labels for preprocessor procedures are necessary for the procedure to be invoked. On a preprocessor procedure, the leading percent sign is only required on the label; statements within the procedure do not require leading percent signs.

The format for a preprocessor label is as follows:

```
%label: preprocessor-statement;
```

For a table summarizing the preprocessor statements and for individual descriptions of the statements, see the *PL/I for OpenVMS Systems Reference Manual*.

Preprocessor Built-In Functions

A number of PL/I preprocessor built-in functions are available for use at compile time. With few exceptions, they have the same effect as run-time PL/I built-in functions with the same name. For a table summarizing the preprocessor built-in functions and for individual descriptions of the functions, see the *PL/I for OpenVMS Systems Reference Manual*.

2.3.4 Compiler Error Messages

One of the functions of the PL/I compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to your default output device; thus, if you enter the PLI command interactively, the messages are displayed on your terminal. If the PLI command is executed in a batch job, the messages appear in the batch job log file.

Each compilation with diagnostic messages terminates with a diagnostic summary that indicates the number of error, warning, and informational messages generated by the compiler. The diagnostic summary has the following format:

```
%PLIG-I-SUMMARY
     Completed with n error(s), n warning(s),
     n informational messages.
```

If the compiler creates a listing file, it also writes the messages to the listing. Messages typically follow the statement that caused the error.

When it appears on the screen, a message from the compiler has the following format:

```
%PLIG-s-ident, message-text At line number n device:[directory]file.ext;x.
```

PL/I for OpenVMS AXP

The PL/I for OpenVMS AXP compiler generally displays the line containing the error also, with an ...^ underneath pointing to the error.

%PLIG

Is the facility, or program, name of the PL/I for OpenVMS VAX compiler. (G denotes the General-Purpose Subset.) This portion indicates that the message is being issued by PL/I.

s

Specifies the severity of the error. Following are the letters that represent the possible severities:

F Fatal. The compiler stops executing, does not continue the compilation, and does not produce an object module. You must correct the error before you can compile the program.

E Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.

W Warning. The compiler produces an object module. It attempts to correct the error in the statement, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.

I Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. Informational messages also indicate nonstandard constructs and items that are syntactically correct, but that may contain programming errors. No action is necessary on your part.

ident
Is the message identification. This gives a descriptive abbreviation of the message text.

message-text
Is the compiler's message. In many cases, the message text consists of more than one line of output. The messages generally provide enough information for you to determine the cause of an error and correct it.

At line number *n*
Specifies the source file line number of the statement that caused the error. This is the line number 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.) Line numbers appear in a listing file.

device:[directory]file.ext;x.
Indicates the file specification.

The compiler produces messages with warning severity if it encounters the following:

- Syntax errors (such as a missing END statement) that the compiler attempts to fix
- Language elements (such as undeclared variables) that are not part of the PL/I General-Purpose Subset but do belong to full PL/I
- Legal PL/I General-Purpose Subset usage (such as assignment of a bit-string value to a fixed-point binary variable) that nonetheless may represent a programming error or produce unexpected results

Most diagnostic messages are self-explanatory; Appendix A lists the diagnostic messages and gives additional explanations.

To examine any diagnostic messages that occurred during the compilation, print the listing file and search for each occurrence of %PLIG.

Section 2.3.5 describes how to read a listing file.

2.3.5 Compiler Listings

Sample annotated listings from the OpenVMS VAX compiler appear in Section 2.3.5.1. Sample annotated listings from the PL/I for OpenVMS AXP appear in Section 2.3.5.2.

2.3.5.1 PL/I for OpenVMS VAX Compiler Listing

The OpenVMS VAX compiler listing displays the following information:

- Effects of the options in the /SHOW qualifier
- The machine code generated by PL/I
- Effects of the /CROSS_REFERENCE qualifier
- Effects of the /OBJECT qualifier

Example 2–1 illustrates the default listing (specified with the /LIST qualifier) and describes the information provided in the listing.

Example 2–2 illustrates a storage map of the program listed in Example 2–1. The PL/I for OpenVMS VAX compiler generates a storage map if you specify /LIST /SHOW=MAP on the PLI command; it also generates a cross-reference listing if you specify /CROSS_REFERENCE.

Example 2–3 illustrates the statistical summary generated if the /LIST /SHOW=STATISTICS qualifier is specified.

Example 2–4 illustrates a portion of a listing of a program compiled with the /LIST/OBJ/MACHINE_CODE qualifiers.

Example 2–5 illustrates the effects of /SHOW=(TRACE,EXPANSION), which shows preprocessor activity in the program listing.

Example 2–1 Default Compiler Listing for VAX Systems

```
FLOWERS 1       2    16-NOV-1991 11:37:00    PL/I for OpenVMS VAX   V3.5-001   3    Page 1
01              4    16-NOV-1991 11:35:49    LI$:[MALCOLM]FLOWERS.PLI;18 (1)

    5
    1 |    /* This procedure obtains data about state flowers from STATEDATA.DAT */
    2
    3
    4           FLOWERS: PROCEDURE OPTIONS(MAIN);
    5    1 6
    6    1      DECLARE EOF BIT(1) STATIC INIT('0'B);
    7    1
    8    1      %INCLUDE 'STATE.TXT';
   23    1
   24    1              ON KEY(STATE_FILE) BEGIN;
   25    2                  PUT SKIP LIST('Error on key',ONKEY(),'error no.',ONCODE());
   26    2                  STOP;
   27    2                  END;
   28    1
   29    1      MODE: BEGIN;
   30    2      DECLARE RUN BIT(1);
   31    2              GET LIST(RUN) OPTIONS(PROMPT('List by state? '));
   32    2              IF RUN THEN GOTO LIST_BY_STATE;
   33    2              GET LIST(RUN) OPTIONS(PROMPT('List by flower? '));
   34    2              IF RUN THEN GOTO LIST_BY_FLOWER;
   35    2              ELSE BEGIN;
   36    3                      DECLARE INPUT_FLOWER CHARACTER(30) VARYING;
   37    3                      GET LIST(INPUT_FLOWER) OPTIONS(PROMPT('Flower? '));
   38    3                      OPEN FILE(STATE_FILE) KEYED ENV(
   39    3                                      INDEX_NUMBER(1),
   40    3                                      SHARED_READ);
   41    3                      READ FILE(STATE_FILE) SET (STATE_PTR) KEY(INPUT_FLOWER);
   42    3                      PUT SKIP EDIT('The flower of',STATE.NAME,'is the',FLOWER)
   43    3                      (3(a));
   44    3              END;
   45    2      END;
   46    1      RETURN;
   47    1      LIST_BY_STATE:
   48    1              ON ENDFILE(STATE_FILE) EOF = '1'B;
   49    1              OPEN FILE(STATE_FILE) SEQUENTIAL ENV(
   50    1                                      INDEX_NUMBER(0),
   51    1                                      SHARED_READ);
   52    1              READ FILE(STATE_FILE) SET (STATE_PTR);
   53    1              DO WHILE (^EOF);
   54    2                  PUT SKIP LIST(STATE.NAME,'flower is ',FLOWER);
   55    2                  READ FILE(STATE_FILE) SET (STATE_PTR);
   56    2              END;
   57    1              CLOSE FILE(STATE_FILE);
   58    1              RETURN;
   59    1       LIST_BY_FLOWER:
   60    1              ON ENDFILE(STATE_FILE) EOF = '1'B;
   61    1              OPEN FILE(STATE_FILE) SEQUENTIAL ENV(
   62    1                                      INDEX_NUMBER(1),
   63    1                                      SHARED_READ);
   64    1              READ FILE(STATE_FILE) SET (STATE_PTR);
   65    1              DO WHILE (^EOF);
   66    2                  PUT SKIP LIST(STATE.NAME,'flower is ',FLOWER);
   67    2                  READ FILE(STATE_FILE) SET (STATE_PTR);
   68    2              END;
   69    1              CLOSE FILE(STATE_FILE);
   70    1              RETURN;
   71    1              END;

 COMMAND LINE
 ------- ----
```

(continued on next page)

Example 2–1 (Cont.) Default Compiler Listing for VAX Systems

```
PLI/LIST FLOWERS 7
```

The following notes are keyed to Example 2–1:

1 The name of the first level-1 procedure in the source program and its identification. If the main procedure did not specify OPTIONS(IDENT), the compiler uses 01 for the identification.

2 The date and time of compilation, and the version of the compiler that was used to compile the program.

3 The page number of the listing file, and the page number of the source file.

4 The date and time that the file containing the source program was created, and its full file specification (to a maximum of 44 characters).

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.

A vertical bar (|) character indicates a line that contains only a comment.

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 The PLI command line as it was entered for compilation.

If the program is compiled with the qualifier /LIST/SHOW=INCLUDE, the %INCLUDE statements are followed by the contents of the INCLUDE files, with line numbers. Notice that INCLUDE files are indicated by an 'I' in the column to the right of the line numbers.

```
 6    1     DECLARE EOF BIT(1) STATIC INIT('0'B);
 7    1
 8    1     %INCLUDE 'STATE.TXT';
 9  I 1     declare 1 state based (state_prt),
10  I 1               2 name character (20),        /* Primary key */
11  I 1               2 population fixed binary(31),/* 3rd alternate key */
12  I 1               2 capital,
13  I 1                 3 name character (20),
14  I 1                 3 population fixed binary(31),
15  I 1               2 largest_cities(2),
16  I 1                 3 name character(30),
17  I 1                 3 population fixed binary(31),
18  I 1               2 symbols,
19  I 1                 3 flower character (30),    /* secondary -- 1st alternate -- key */
20  I 1                 3 bird character (30),      /* tertiary -- 2nd alternate -- key */
21  I 1             state_ptr pointer,
22  I 1             state_file file;
23    1
24    1             ON KEY(STATE_FILE) BEGIN;
25    2                 PUT SKIP LIST('Error on key',ONKEY(),'error no.',ONCODE());
26    2                 STOP;
27    2                 END;
28    1
```

Example 2–2 illustrates the storage map page of the program listing. This page is generated if /LIST/SHOW=MAP is specified on the PLI command.

Example 2–2 Compiler Storage Map for VAX Systems

```
FLOWERS                  12-MAR-1991 11:36:16    PL/I for OpenVMS VAX   V3.5-001   Page 2
01                       16-MAR-1991 11:35:49    LI$:[MALCOLM]FLOWERS.PLI;17 (1)

                               +-------------+
                               | Storage Map |
                               +-------------+

External Entry Points and Variables Declared Outside Procedures  1
------------------------------------------------------------

 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

FLOWERS                                      4     ENTRY, EXTERNAL

Procedure FLOWERS on line 3
--------------------------

 Identifier Name 2     Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

BIRD                              30 BY    22    OFFSET FROM BASE IS 146 BY, MEMBER OF STRUCTURE
                                                    SYMBOLS CHARACTER(30), UNALIGNED

CAPITAL                           24 BY    22    OFFSET FROM BASE IS 24 BY, MEMBER OF STRUCTURE
                                                    STATE,STRUCTURE

EOF                    static     1 BI     6     BIT(1), UNALIGNED, INITIAL, INTERNAL

FLOWER                            30 BY    22    OFFSET FROM BASE IS 116 BY, MEMBER OF STRUCTURE
                                                    SYMBOLS CHARACTER(30), UNALIGNED,

LARGEST_CITIES                    68 BY    22    OFFSET FROM BASE IS 48 BY, MEMBER OF STRUCTURE
                                                    STATE, STRUCTURE DIMENSION
   .
   .
   .
Begin Block on line 29
----------------------

 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

RUN                    automatic  1 BI     30    BIT(1), UNALIGNED

Begin Block on line 35
----------------------

 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

INPUT_FLOWER           automatic  32 BY    36    CHARACTER(30), VARYING, UNALIGNED

 Psect Name  3        Allocation  Attributes
 ----- ----          ----------  ----------

 $CODE               1221 by     position-independent, relocatable, share, execute, read
 $DATA               1 by        position-independent, relocatable, read, write
 $ADDRESS_DATA       0 by        position-independent, relocatable, read
 SYSIN               450 by      position-independent, overlay, relocatable, global, read, write
 SYSPRINT            450 by      position-independent, overlay, relocatable, global, read, write
 STATE_FILE          451 by      position-independent, overlay, relocatable, global, read, write
```

(continued on next page)

Example 2–2 (Cont.) Compiler Storage Map for VAX Systems

```
Procedure Definition Map 4
--------- ---------- ---

 Line    Name
 ----    ----

 5       FLOWERS
 26       BEGIN
 31       BEGIN
 37        BEGIN

 COMMAND LINE 5
 ------- ----

PLI/LIST/SHOW=MAP FLOWERS
```

The following notes are keyed to Example 2–2:

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 and *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 continues 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.

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.

When PLI/LIST/SHOW=MAP/CROSS_REFERENCE is specified, each name that is referenced is followed by a list of the numbers of all lines that contain references to that name. For example:

```
 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

BIRD                              30 BY    22    OFFSET FROM BASE IS 146 BY, MEMBER OF STRUCTURE
                                                    SYMBOLS CHARACTER(30), UNALIGNED
                                                    No references.

CAPITAL                           24 BY    22    OFFSET FROM BASE IS 24 BY, MEMBER OF STRUCTURE
                                                    STATE, STRUCTURE
                                                    No references.

EOF                    static     1 BI     6     BIT(1), UNALIGNED, INITIAL, INTERNAL
                                                    Reference lines: 48, 53, 56, 60, 65, 68
```

```
FLOWER                          30 BY    22    OFFSET FROM BASE IS 116 BY, MEMBER OF STRUCTURE
                                                        SYMBOLS CHARACTER(30), UNALIGNED,
                                                        Reference lines: 43, 54, 66

Begin Block on line 29
----------------------

 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

RUN                    automatic   1 BI     30    BIT(1), UNALIGNED
                                                        Reference lines: 31, 32, 33, 34

Begin Block on line 35
----------------------

 Identifier Name       Storage     Size     Line  Attributes
 ---------- ----       -------     ----     ----  ----------

INPUT_FLOWER           automatic   32 BY    36    CHARACTER(30), VARYING, UNALIGNED
                                                        Reference lines: 37, 41
```

Example 2–3 illustrates the statistical summary that PL/I includes in the listing if the /LIST/SHOW=STATISTICS qualifier is specified. The following notes are keyed to Example 2–3:

1 The compiler accumulates statistics for each phase of its operation.

2 For each phase of its operation, the compiler lists I/O, memory, and CPU time usage statistics.

Example 2–3 Compiler Performance Statistics for VAX Systems

```
FLOWERS                    22-MAR-1991 09:59:16    PL/I for OpenVMS VAX   V3.5-001            Page 3
01                         16-MAR-1991 11:35:49    LI$:[MALCOLM]FLOWERS.PLI;18 (1)

                                      +-----------------------+
                                      | Performance Indicators |
                                      +-----------------------+
 1                        2
phase                     buf i/o  dir i/o pageflt virtmem workset cputim
-----                     -------  ------- ------- ------- ------- ------
pass 1 totals                 4      34     116       0    1350      78
declare totals                0       0       4       0    1500       6
pass 2 totals                 0       0      64       0    1800      41
  live analysis               0       0      29       0    2048       7
  reorder invariants          0       0      30       0    2048       3
  eliminate redundancy        0       0      12       0    2048       9
optimizer totals              0       0      98       0    2048      32
allocator totals              1       3      28       0    2048      13
  generate code list          0       0      26       0    2048      35
  register allocation         0       0       0       0    2048       6
  peephole optimization       0       0       3       0    2048      13
  branch/jump resolution      0       0       0       0    2048       2
  write object module         0       0       2       0    2048       8
code generator totals         0       0      34       0    2048      66
total compilation             8      45     827     432    2048     300
71 lines compiled
compilation rate was 1420 lines per minute
```

If you specify /LIST/OBJECT/MACHINE_CODE when you compile a PL/I program, the compiler includes the generated assembly language code and object code in the listing. Example 2–4 illustrates this listing.

Example 2–4 Machine Code Listing for VAX Systems

```
FLOWERS                 22-MAR-1991 10:00:37    PL/I for OpenVMS VAX   V3.5-001             Page 1
01                      16-MAR-1991 11:35:49    LI$:[MALCOLM]FLOWERS.PLI;18 (1)


    1 |    /* This procedure obtains data about state flowers from STATEDATA.DAT */
    2
    3
    4           FLOWERS: PROCEDURE OPTIONS(MAIN);
                                        0073  FLOWERS:
                                    C00C  0073          .entry  FLOWERS,^m<dv,iv,r2,r3>
                                5E 10 C2  0075          subl2   #10,sp
                              00000000* EF 16  0078          jsb     PLI$OPTIONSMAIN
                            5C 00000000 EF 9E  007E          movab   STATE_FILE,ap
                            52 00000000 EF 9E  0085          movab   $DATA,r2
                                53 5E D0  008C          movl    sp,r3
    5    1
    6    1      DECLARE EOF BIT(1) STATIC INIT('0'B);
    7    1
    8    1      %INCLUDE 'STATE.TXT';
   23    1
   24    1              ON KEY(STATE_FILE) BEGIN; 1
                            5E 53 14 C3  008F subl3   #14,r3,sp 2
                              53 5E D0  0093          movl    sp,r3
                            0C AE 6C 9E  0096          movab   (ap),0C(sp)
                      04 AE 001E8024 8F D0  009A          movl    #1E8024,04(sp)
                            08 AE 0D AF 9E  00A2          movab   sym.1,08(sp)
                              6E F4 AD D0  00A7          movl    -0C(fp),(sp)
                              F4 AD 5E D0  00AB          movl    sp,-0C(fp)
                                  0092 31  00AF          brw     sym.4
                                        00B2  sym.1:
                                    C000  00B2          .entry  vcg.code,^m<dv,iv>
   25    2                PUT SKIP LIST('Error on key',ONKEY(),'error no.',ONCODE());
                              51 5D D0  00B4          movl    fp,r1
                            02 AF 6C FA  00B7          callg   (ap),sym.2
                                 7D 11  00BB          brb     sym.3
                                        00BD  sym.2:
                                    C87C  00BD          .entry  vcg.code,^m<dv,iv,r2,r3,r4,r5,r6,r11>
                             5E FEF8 CE 9E  00BF          movab   -0108(sp),sp
                          5C 00000000 EF 9E  00C4          movab   SYSPRINT,ap
                                      50 7C  00CB          clrq    r0
                                FE AD 01 B0  00CD          movw    #1,-02(fp)
                                52 FE AD 9E  00D1          movab   -02(fp),r2
                                      53 7C  00D5          clrq    r3
                              00000000* EF 16  00D7          jsb     PLI$PUTFILE_R6
                                 50 81 AF 9E  00DD          movab   $CODE+61,r0
                                    51 0C 3C  00E1          movzwl  #C,r1
                              00000000* EF 16  00E4          jsb     PLI$PUTLCHAR_R6
                                   FF7E CD 9F  00EA          pushab  -0082(fp)
                                7E 80 8F 9A  00EE          movzbl  #80,-(sp)
                           00000000* EF 02 FB  00F2          calls   #2,PLI$ONKEY
                                52 FF7E CD B0  00F9          movw    -0082(fp),r2
                                50 FF7E CD 9E  00FE          movab   -0082(fp),r0
                                    51 52 3C  0103          movzwl  r2,r1
                              00000000* EF 16  0106          jsb     PLI$PUTLVCHA_R6
                                50 FF48 CF 9E  010C          movab   $CODE+58,r0
                                    51 09 3C  0111          movzwl  #9,r1
                              00000000* EF 16  0114          jsb     PLI$PUTLCHAR_R6
```

The following notes are keyed to Example 2–4:

1 The machine 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 code. To the left of the object location is the object code generated by the PL/I for OpenVMS VAX compiler.

If you specify /LIST/SHOW=(INCLUDE,EXPANSION,TRACE) when you compile a program that uses the embedded preprocessor, the compiler includes additional preprocessor information in the listing. Example 2–5 illustrates some of the notations that can appear in the column to the right of the line numbers in the listing. Except as indicated, the notations seen here are enabled by default.

Example 2–5 Preprocessor Compiler Listing for VAX Systems

```
WR_SCHEDULE              12-MAR-1991 16:00:27    PL/I for OpenVMS VAX   V3.5-001             Page 1
01                       12-MAR-1991 15:57:14    APLD$:[MALCOLM]SCHEDULE.PLI;1 (1)


    1          WR_SCHEDULE:  PROCEDURE OPTIONS (MAIN);
    2    1                   DECLARE I FIXED BINARY(15);
    3    1                   %DECLARE (DOWN_PAGE,ACROSS_PAGE) CHAR;
    4    1
    5    1                   %ACROSS_PAGE = 'PAGE(ACROSS)';
    6    1                      %DOWN_PAGE   = 'PAGE(DOWN)';
    7    1                      %IF VARIANT() = 'DAYS' %THEN %INCLUDE 'DAYS.PLI';
1   8 |I 1    /*  THE INCLUDE FILE 'DAYS.PLI' CREATES AN ARRAY STRUCTURE DAYS AND     */
    9 |I 1 /*  INITIALIZES IT WITH THE DAYS OF THE WEEK.                          */
   10  I 1
   11  I 1     %DO;
   12  I 1
   13  I 1        %DECLARE SCHEDULE CHAR,NUM_ITEMS FIXED;
   14  I 1           %SCHEDULE = 'DAYS';

2  15  I 1       %NUM_ITEMS = 7;
   16  I 1
   17  I 1           DECLARE DAYS(7) CHAR(10) INIT(
   18  I 1             'MONDAY','TUESDAY','WEDNESDAY','THURSDAY','FRIDAY',
   19  I 1             'SATURDAY','SUNDAY');
   20  I 1     %END;
3  21 *  1                    %ELSE %IF VARIANT() = 'MONTHS' %THEN %INCLUDE 'MONTHS.PLI';
   22    1
   23 *  1                     %WRITE:  PROCEDURE(SCHEDULE,PAGE) STATEMENT RETURNS(CHAR);
   24 P  1                              DECLARE (SCHEDULE,PAGE,F_SPEC) CHAR;
   25    1
   26    1                                IF PAGE = 'DOWN'
   27 P  1                                  THEN DO;
4  28 P  1                                 F_SPEC = ' DO I = 1 TO NUM_ITEMS)) (A(10),SKIP);';
   29 P  1                               RETURN ('PUT SKIP EDIT(('||SCHEDULE||'(I)'||F_SPEC);
   30 P  1                                       END;
   31    1                                  ELSE IF PAGE = 'ACROSS'
   32 P  1                                        THEN RETURN ('PUT SKIP LIST ('||SCHEDULE||');');
   33 P  1                                  END;
   34    1
   35    1                           WRITE(SCHEDULE) DOWN_PAGE;
5     T                              WRITE(DAYS) DOWN_PAGE;
      T                              WRITE(DAYS) PAGE(DOWN);
      T                              PUT SKIP EDIT((DAYS(I) DO I = 1 TO 7)) (A(10),SKIP);
6     E                              PUT SKIP EDIT((DAYS(I) DO I = 1 TO 7)) (A(10),SKIP);
   36    1                           WRITE(SCHEDULE) ACROSS_PAGE;
      T                              WRITE(DAYS) ACROSS_PAGE;
      T                              WRITE(DAYS) PAGE(ACROSS);
      E                              PUT SKIP LIST (DAYS);
   37    1                   END;


 COMMAND LINE
 ------- ----

PLI/LIST/SHOW=(INCLUDE,EXPAN,TRACE)/VARIANT=DAYS SCHEDULE
```

The following notes are keyed to Example 2–5:

1 The operand symbol (|) denotes a line of source text that contains comment text.

2 The I indicates text from an INCLUDE file. /SHOW=INCLUDE enables this indicator.

3 The asterisk (*) indicates unused preprocessor text.

4 The P indicates lines contained within a preprocessor procedure.

5 The T indicates each instance of preprocessor variable value replacement. /SHOW=TRACE enables this indicator.

6 The E indicates the final replacement value for the preprocessor variable. /SHOW=EXPANSION enables this indicator.

2.3.5.2 PL/I for OpenVMS AXP Compiler Listing

The PL/I for OpenVMS AXP compiler listing displays the the following information:

Example 2–6 Default Compiler Listing for AXP Systems

```
FLOWERS 1              Source Listing 2  19-AUG-1993 10:26:27    DEC PL/I V4.0-001  3  Page   1
01                                    4  19-AUG-1993 10:25:07    DISK$DISK4:[MAZORA.DECPLI.LZ]FLOWERS.PLI;1

  |    5     1 /* This procedure obtains data about state flowers from STATEDATA.DAT */
           2
           3
     6     4 FLOWERS: PROCEDURE OPTIONS(MAIN);
     1     5
     1     6 DECLARE EOF BIT(1) STATIC INIT('0'B);
     1     7
     1     8 %INCLUDE 'STATE.TXT';
     1    23
     1    24  ON KEY(STATE_FILE) BEGIN;
     1    25      PUT SKIP LIST('Error on key',ONKEY(),'error no.',ONCODE());
     1    26      STOP;
     1    27      END;
     1    28
     1    29 MODE: BEGIN;
     1    30 DECLARE RUN BIT(1);
     1    31         GET LIST(RUN) OPTIONS(PROMPT('List by state? '));
     1    32         IF RUN THEN GOTO LIST_BY_STATE;
     1    33         GET LIST(RUN) OPTIONS(PROMPT('List by flower? '));
     1    34         IF RUN THEN GOTO LIST_BY_FLOWER;
     1    35         ELSE BEGIN;
     1    36                 DECLARE INPUT_FLOWER CHARACTER(30) VARYING;
     1    37                 GET LIST(INPUT_FLOWER) OPTIONS(PROMPT('Flower? '));
     1    38                 OPEN FILE(STATE_FILE) KEYED ENV(
     1    39                                 INDEX_NUMBER(1),
     1    40                                 SHARED_READ);
     1    41                 READ FILE(STATE_FILE) SET (STATE_PTR) KEY(INPUT_FLOWER);
     1    42                 PUT SKIP EDIT('The flower of',STATE.NAME,'is the',FLOWER)
     1    43                 (3(a));
     1    44         END;
     1    45         END;
     1    46         RETURN;
     1    47         LIST_BY_STATE:
     1    48                 ON ENDFILE(STATE_FILE) EOF = '1'B;
     1    49                 OPEN FILE(STATE_FILE) SEQUENTIAL ENV(
     1    50                 INDEX_NUMBER(0),
     1    51                 SHARED_READ);
     1    52         READ FILE(STATE_FILE) SET (STATE_PTR);
     1    53         DO WHILE (^EOF);
     1    54             PUT SKIP LIST(STATE.NAME,'flower is ',FLOWER);
     1    55             READ FILE(STATE_FILE) SET (STATE_PTR);
     1    56             END;
     1    57             CLOSE FILE(STATE_FILE);
     1    58             RETURN;
     1    59 LIST_BY_FLOWER:
     1    60         ON ENDFILE(STATE_FILE) EOF = '1'B;
     1    61         OPEN FILE(STATE_FILE) SEQUENTIAL ENV(
     1    62                                 INDEX_NUMBER(1),
     1    63                                 SHARED_READ);
     1    64         READ FILE(STATE_FILE) SET (STATE_PTR);
     1    65         DO WHILE (^EOF);
     1    66             PUT SKIP LIST(STATE.NAME,'flower is ',FLOWER);
     1    67             READ FILE(STATE_FILE) SET (STATE_PTR);
     1    68             END;
     1    69         CLOSE FILE(STATE_FILE);
     1    70         RETURN;
     1    71         END;

 COMMAND LINE
 ------- ----

PLI/LIST FLOWERS 7
```

The following notes are keyed to Example 2–6:

1 The name of the first level-1 procedure in the source program and its identification. If the main procedure did not specify OPTIONS(IDENT), the compiler uses 01 for the identification.

2 The date and time of compilation, and the version of the compiler that was used to compile the program.

3 The page number of the listing file, and the page number of the source file.

4 The date and time that the file containing the source program was created, and its full file specification (to a maximum of 44 characters).

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.

A vertical bar (|) character indicates a line that contains only a comment.

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 The PLI command line as it was entered for compilation.

If the program is compiled with the qualifier /LIST/SHOW=INCLUDE, the %INCLUDE statements are followed by the contents of the INCLUDE files, with line numbers. Notice that INCLUDE files are indicated by an 'I' in the column to the right of the line numbers.

Example 2–7 Compiler Storage Map for AXP Systems

```
FLOWERS                          Source Listing                 19-AUG-1993 10:44:46    DEC PL/I V4.0-001                  Page
01                                                              19-AUG-1993 10:25:07    DISK$DISK4:[MAZORA.DECPLI.LZ]FLOWERS.PLI;1

          +-------------+
          | Storage Map |
          +-------------+

External Entry Points and Variables Declared Outside Procedures 1
-----------------------------------------------------------------

 Identifier Name 2                Storage     Size     Line  Attributes
 ---------- ----                  -------     ----     ----  ----------

 FLOWERS                                              4     ENTRY, EXTERNAL
Procedure FLOWERS on line 4
---------------------------

 Identifier Name e                Storage     Size     Line  Attributes
 ---------- ----                  -------     ----     ----  ----------

 BIRD                                         30 BY    20    OFFSET FROM BASE IS 146 BY, MEMBER OF STRUCTURE SYMBOLS, CHARACTER(30)
                                                            UNALIGNED, NONVARYING

 CAPITAL                                      24 BY    12    OFFSET FROM BASE IS 24 BY, MEMBER OF STRUCTURE STATE, STRUCTURE

 EOF                              static      1 BI     6     BIT(1), UNALIGNED, INTERNAL, INITIAL, NONVARYING

 FLOWER                                       30 BY    19    OFFSET FROM BASE IS 116 BY, MEMBER OF STRUCTURE SYMBOLS, CHARACTER(30)
                                                            UNALIGNED, NONVARYING

 LARGEST_CITIES                               68 BY    15    OFFSET FROM BASE IS 48 BY, MEMBER OF STRUCTURE STATE, STRUCTURE
                                                            DIMENSION

   .
   .
   .
Begin Block on line 29
----------------------

 Identifier Name                  Storage     Size     Line  Attributes
 ---------- ----                  -------     ----     ----  ----------

 RUN                              automatic   1 BI     30    BIT(1), UNALIGNED, NONVARYING

Begin Block on line 35
----------------------

 Identifier Name                  Storage     Size     Line  Attributes
 ---------- ----                  -------     ----     ----  ----------

 INPUT_FLOWER                     automatic   32 BY    36    CHARACTER(30), VARYING, UNALIGNED

Procedure Definition Map 3
--------- ---------- ---

 Line    Name
 ----    ----

 4  FLOWERS
 24   BEGIN
 29   BEGIN
 35    BEGIN

 COMMAND LINE 4
 ------- ----

PLI/LIST/SHOW=MAP FLOWERS
```

The following notes are keyed to Example 2–7:

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 and *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 continues 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 Procedure Definition Map lists each procedure and begin block in the program, giving the line number on which the block is defined.

4 The Command Line shows the PLI command string that was processed, including input files, qualifiers, and library files.

Example 2–8 Compiler Performance Statistics for AXP Systems

```
FLOWERS                         Compilation statistics          19-AUG-1993 10:56:56    DEC PL/I V4.0-001                  Page
01                                                              19-AUG-1993 10:25:07    DISK$DISK4:[MAZORA.DECPLI.LZ]FLOWERS.PLI;1

COMPILER INTERNAL TIMING

       Phase  1                        CPU 2     Elapsed    Page      I/O
                                    seconds  seconds   faults    count
  pass 1 totals                        0.08     1.63       52        5
  declare totals                       0.01     0.62        3        0
  pass 2 totals                        0.09     1.34       15        0
  translation totals                   0.04     2.62       58        0
  Optimization                         0.19     0.76        5        0
    Dominator tree construction        0.00     0.00        0        0
    Loop dominator insertion           0.00     0.00        0        0
    Lifetime analysis                  0.00     0.00        0        0
    IDEF computation                   0.01     0.00        0        0
    DATAFLOW computation               0.06     0.19        1        0
    Strength reduction                 0.02     0.00        0        0
    Loop unroll/Loop optimization      0.00     0.00        0        0
    Test replacement                   0.01     0.00        0        0
    Profitability determination        0.02     0.29        2        0
    Profitability reordering           0.00     0.00        0        0
    Use propagation                    0.00     0.00        0        0
    Split lifetime analysis            0.02     0.03        0        0
  Code Generation                      0.36     2.66       47        0
    Context analysis                   0.16     1.80       17        0
    Register History                   0.03     0.25        2        0
    Temporary allocation               0.05     0.35        3        0
    Code emission                      0.07     0.23       18        0
  Final                                0.13     0.12        0        0
    Peepholing                         0.03     0.01        0        0
    Object scheduling                  0.08     0.11        0        0
    Branch/jump resolution             0.01     0.00        0        0
  Object module generation             0.00     0.67        1       10
  Compiler totals                      0.98    15.49      253       20
COMPILATION STATISTICS

  CPU time:          0.98 seconds
  Elapsed time:     15.49 seconds
  Pagefaults:         253
  I/O Count:           20
  Source lines:        71

  4346 lines per CPU minute.
```

If you specify /LIST/OBJECT/MACHINE_CODE when you compile a PL/I program, the compiler includes the generated assembly language code and object code in the listing. Example 2–9 illustrates this listing.

Example 2–9 Machine Code Listing for AXP Systems

```
FLOWERS                         Machine Code Listing           19-AUG-1993 10:59:36    DEC PL/I V4.0-001                  P
01                              FLOWERS                        19-AUG-1993 10:25:07    DISK$DISK4:[MAZORA.DECPLI.LZ]FLOWERS.

    .PSECT $CODE$, OCTA, PIC, CON, REL, LCL, SHR,-
     EXE, NORD, NOWRT
      0000 FLOWERS::              ; 000004
23DEFF10     0000  LDA SP, -240(SP)    ; SP, -240(SP) 1
47FF0419     0004  CLR R25     ; R25
B7FE00A0     0008  STQ R31, 160(SP)    ; R31, 160(SP)
B77E0000     000C  STQ R27, (SP)    ; R27, (SP)
B75E00B8     0010  STQ R26, 184(SP)    ; R26, 184(SP)
B45E00C0     0014  STQ R2, 192(SP)    ; R2, 192(SP)
B47E00C8     0018  STQ R3, 200(SP)    ; R3, 200(SP)
B49E00D0     001C  STQ R4, 208(SP)    ; R4, 208(SP)
B4BE00D8     0020  STQ R5, 216(SP)    ; R5, 216(SP)
B7BE00E0     0024  STQ FP, 224(SP)    ; FP, 224(SP)
63FF0000     0028  TRAPB      ;
47FE041D     002C  MOV SP, FP     ; SP, FP
47FB0402     0030  MOV R27, R2     ; R27, R2
23DEFF80     0034  LDA SP, -128(SP)    ; SP, -128(SP)
A742FFC8     0038  LDQ R26, -56(R2)    ; R26, -56(R2)
A762FFD0     003C  LDQ R27, -48(R2)    ; R27, -48(R2)
6B5A4000     0040  JSR R26, DPLI$HND_OPTIONS_MAIN  ; R26, R26
263F001F     0044  LDAH R17, 31(R31)    ; R17, 31(R31)        ; 000024
22318024     0048  LDA R17, -32732(R17)   ; R17, -32732(R17)
A402FEE8     004C  LDQ R0, -280(R2)    ; R0, -280(R2)
B3FD0090     0050  STL R31, 144(FP)    ; R31, 144(FP)
221D0010     0054  LDA R16, 16(FP)    ; R16, 16(FP)
B23D0098     0058  STL R17, 152(FP)    ; R17, 152(FP)

   .
   .
   .

Routine Size: 1616 bytes,    Routine Base: $CODE$ + 0000
```

The following notes are keyed to Example 2–9:

1 The machine 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 code. To the left of the object location is the object code generated by the PL/I for OpenVMS VAX compiler.

2.4 Linking a PL/I Program

Once you have compiled a PL/I source program or module, use the DCL command LINK to combine your object modules into one executable image, which can then be executed by the OpenVMS system. A source program or module cannot run on the OpenVMS system until it is linked.

When you execute the LINK command, the OpenVMS Linker performs the following functions:

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

When linking on development systems, you may want to use the /DEBUG qualifier. The /DEBUG qualifier appends to the image all the symbol and line-number information appended to the object modules, plus information on global symbols, and causes the image to run under debugger control when it is executed.

The LINK command produces an executable image by default. However, you can also use the LINK command to obtain shareable images and system images. Section 2.4.2 describes LINK command qualifiers.

For a complete discussion of the OpenVMS Linker, see the *OpenVMS Linker Utility Manual*.

2.4.1 LINK Command

The LINK command has the following format:

LINK[/command-qualifier]... {file-spec[/file-qualifier...]},...

/command-qualifier...
Specifies output file options.

file-spec
Specifies the input files to be linked.

/file-qualifier...
Specifies input file options.

If you specify more than one input file, you must separate the input file specifications with a plus sign (+) or a comma (,).

By default, the linker creates an output file with the name of the first input file specified and the file type EXE. Therefore, when you link more than one file, it is good practice to list the file containing the main program first so that the name of your output file will have the same name as that of your main program module.

The following command line links the object files MAINPROG.OBJ, SUBPROG1.OBJ, and SUBPROG2.OBJ to produce one executable image called MAINPROG.EXE.

```
$ LINK MAINPROG.OBJ, SUBPROG1.OBJ, SUBPROG2.OBJ
```

2.4.2 LINK Command Qualifiers

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

The following list summarizes some of the most commonly used LINK command qualifiers. A brief description of each qualifier follows this list. For a complete list of LINK qualifiers, see the *OpenVMS Linker Utility Manual*.

Command Qualifiers	Default
/[NO]EXECUTABLE[=file-spec]	/EXECUTABLE=name.EXE
/[NO]SHAREABLE[=file-spec]	/NOSHAREABLE
/BRIEF	
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/FULL	
/[NO]MAP	/NOMAP (interactive)
/[NO]DEBUG	/NODEBUG
/[NO]TRACEBACK	/TRACEBACK

/EXECUTABLE [=file-spec]
/NOEXECUTABLE
Causes the linker to produce or to suppress the production of an executable image.

/SHAREABLE [=file-spec]
/NOSHAREABLE (default)
Causes the linker to create or not create a shareable image.

/BRIEF
Causes the linker to produce a summary of the image's characteristics and a list of contributing modules.

/CROSS_REFERENCE
/NOCROSS_REFERENCE (default)
Causes the linker to produce cross-reference information for global symbols or to suppress its creation.

/FULL
Causes the linker to produce a summary of the image's characteristics, a list of contributing modules, listings of global symbols by name and by value, and a summary of characteristics of image sections in the linked image.

/MAP
/NOMAP (interactive default)
Causes the linker to generate or not generate a map file.

/DEBUG
/NODEBUG (default)
Causes the linker to include or not include the VMS Debugger in the executable image and generate or not generate a symbol table.

/TRACEBACK (default)
/NOTRACEBACK
Causes the linker to generate symbolic traceback information when error messages are produced or to suppress its generation.

2.4.3 Linker Input Files

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

- Specify input file specifications for the object modules.

  If no file type is specified, the linker searches for an object file with the file type OBJ.

- Specify one or more object module library files.

  You can specify either the name of an object module library with the /LIBRARY qualifier or the names of object modules contained in an object module library with the /INCLUDE qualifier. The uses of object module libraries are described in Section 2.4.5.

- Specify an options file.

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

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

File Type	File
OBJ	Object module
OLB	Library
OPT	Options file

2.4.4 Linker Output Files

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

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

To specify an alternative name for a map file or image file or to specify an alternative output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. In the following example, the LINK command creates the image file [PROJECT.EXE]UPDATE.EXE and the map file [PROJECT.MAP]UPDATE.MAP:

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

2.4.5 Object Module Libraries

You can make program modules accessible to other users by storing them in an object module library. To link modules contained in an object module library, use the /INCLUDE qualifier and specify the modules you want to link. In the following example, the LINK command directs the linker to link the subprogram modules EGGPLANT, TOMATO, BROCCOLI, and ONION with the main program module GARDEN:

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

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

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

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

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

For more information about object module libraries, see the *OpenVMS Linker Utility Manual*.

2.4.6 Linker Error Messages

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

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

- An object module has compilation errors.

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

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

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

- You tried to link a nonexistent module.

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

- A reference to a symbol name remains unresolved.

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

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

  Because SEAWEED is not linked, the linker signals the following error messages:

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

If an error occurs when you link modules, you can often correct the error by reissuing the command string and specifying the correct modules or libraries. If an error indicates that a program module cannot be located, you may be linking the program with the wrong PL/I Run-Time Library.

For a complete list of linker messages, from both the OpenVMS VAX and PL/I for OpenVMS AXP compilers, see *OpenVMS System Messages and Recovery Procedures Reference Manual*.

2.5 Running a PL/I Program

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

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

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

file-spec
Specifies the file you want to run.

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

```
$ RUN SAMPLE/NODEBUG
```

See Chapter 3 for more information on debugging programs.

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

For example, if an integer divide-by-zero condition occurs and if 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_F, arithmetic fault, floating divide
by zero at PC=000007C4, PSL=03C000A5
```

These messages are followed by a traceback message like the following:

```
%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
```

module name
Indicates the name of a level-1 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 named on the previous line. In this example, the level-1 procedures are LIBS and SETUP; a call to SETUP occurred during the execution of LIBS.

routine name
Indicates the entry name of the internal procedure or block in the calling sequence. When BEGIN%n appears in this column, it indicates that an unlabeled begin block, a PUT statement, or a GET statement was active when the error occurred.

PL/I assigns labels to these blocks, giving them names in this form, where n is the source program line number on which the block is entered.

In this example, an unlabeled begin block or PUT or GET statement occurs on line 4 of the routine SETUP; within this block or statement, the routine DIVIDE was invoked. Thus, this traceback indicates that the error occurred during execution of an instruction generated for the source statement on line 9, in the procedure DIVIDE.

line
Indicates the source program line number (generated by the compiler) of the statement at which the error occurred, or at which the call or reference to the next procedure was made. This line number matches the line numbers on the listing file created if /LIST was specified to the compiler.

relative PC
Gives the value of the PC (program counter). This value represents the location in the program image at which the error occurred or at which a procedure was called. The location is relative to the virtual memory address that the linker assigned to the code program section of the module indicated by module name.

absolute PC
Gives the value of the PC in absolute terms, that is, the actual address in virtual memory that represents the location at which the error occurred.

Traceback information is available at run time only for modules that were compiled and linked with the traceback option in effect. The traceback option is in effect by default for both the PLI and LINK commands. You can use the PLI command qualifier /NODEBUG and the LINK command qualifier /NOTRACEBACK to exclude traceback information. However, it is recommended that you exclude traceback information only from thoroughly debugged program modules.

For a complete list of PL/I for OpenVMS VAX run-time error messages, see Appendix A.

3
Using the VMS Debugger

This chapter is an introduction to using the VMS Debugger with PL/I for OpenVMS VAX programs. This chapter provides the following information:

- An overview of the debugger
- Information to get you started using the debugger
- A sample terminal session that demonstrates using the debugger
- A list of the debugger commands by function

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

3.1 Overview

A debugger is a tool that helps you locate run-time errors quickly. It is used with a program that has already been compiled and linked successfully, but does not run correctly. For example, the output may be obviously wrong, or the program goes into an infinite loop or terminates prematurely. The debugger enables you to observe and manipulate the program's execution interactively so that you can locate the point at which the program stopped working correctly.

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

The debugger recognizes the syntax, expressions, data typing, and other constructs of PL/I for OpenVMS VAX, as well as other VAX languages including:

VAX Ada
VAX BASIC
VAX BLISS
VAX C
VAX COBOL
VAX DIBOL
VAX FORTRAN
VAX MACRO-32
VAX Pascal
VAX RPG II
VAX SCAN

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

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

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

Such techniques allow you to isolate an error in your code much more quickly than you could without the debugger.

Once you have found the error in the program, you can then edit the source code and compile, link, and run the corrected version.

3.2 Features of the Debugger

The VMS Debugger provides the following features to help you debug your programs:

- **Online HELP**

  Online HELP is always available during a debugging session and contains information on all the debugger commands and also information on selected topics.

- **Source Code Display**

  You can display lines of source code during a debugging session.

- **Screen Mode**

  You can capture and display various kinds of information in scrollable windows, which can be moved around the screen and resized. Automatically updated source, instruction, and register displays are available. You can selectively direct debugger input, output, and diagnostic messages to displays. (Screen mode displays work best on VT100-series or VT200-series terminals or MicroVAX workstations.)

- **Keypad Mode**

  When you invoke the debugger, several commonly used debugger command sequences are assigned by default to the keys of the numeric keypad (if you have a VT100, VT52, or LK201 keypad).

- **Source Editing**

  As you find errors during a debugging session, you can use the EDIT command to invoke any editor available on your system. (You specify the editor you want with the SET EDITOR command.)

- **Command Procedures**

  The debugger allows you to execute a command procedure to recreate a debugging session, to continue a previous session, or to avoid typing the same debugger commands many times during a debugging session.

- **Symbol Definitions**

  You can define your own symbols to represent lengthy commands, address expressions, or values.

- **Initialization Files**

  You can create an initialization file containing commands to set your default debugging modes, screen display definitions, keypad key definitions, symbol definitions, and so on. In addition, you may want to have special initialization files for debugging specific programs.

- **Log Files**

  You can record the commands you issue during a debugging session and the debugger's responses to those commands in a log file. You can use log files to keep track of your debugging efforts, or you can use them as command procedures in subsequent debugging sessions.

3.3 Getting Started with the Debugger

This section explains how to use the debugger with PL/I for OpenVMS VAX programs. The section focuses on basic debugger functions, to get you started quickly. It also provides any debugger information that is specific to PL/I for OpenVMS VAX. For more detailed information that is not specific to a particular language, see the *OpenVMS Debugger Manual*.

3.3.1 Compiling and Linking a Program to Prepare for Debugging

Before you can use the debugger, you must compile and link your program as explained in this section. The following example shows how to compile and link a PL/I for OpenVMS VAX program (consisting of a single compilation unit named INVENTORY) to prepare for using the debugger.

```
$ PLI/DEBUG/NOOPTIMIZE INVENTORY
$ LINK/DEBUG INVENTORY
```

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

You should use the /NOOPTIMIZE qualifier when you compile a program in preparation for debugging. Otherwise, if the object code is optimized (to reduce the size of the program and make it run faster), the contents of some program locations may be inconsistent with what you might expect from viewing the source code. After debugging the program, you should recompile it without the /NOOPTIMIZE qualifier.

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

3.3.2 Starting and Terminating a Debugging Session

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

```
$ RUN INVENTORY

                    VAX DEBUG Version <VMS_VERSION>

%DEBUG-I-INITIAL, language is PL/I for OpenVMS VAX, module set to 'INVENTORY'
DBG>
```

The INITIAL message indicates that the debugging session is initialized for a PL/I for OpenVMS VAX program and that the name of the main program unit is INVENTORY. The DBG> prompt indicates that you can now type debugger commands. At this point, if you type the GO command, program execution begins and continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

If you have a mixed-language program that includes an Ada package, the following message will appear on your screen instead of the previous one when you invoke the debugger:

```
$ RUN INVENTORY

                    VAX DEBUG Version <VMS_VERSION>

%DEBUG-I-INITIAL, language is PL/I for OpenVMS VAX, module set to 'INVENTORY'
%DEBUG-I-NOTATMAIN, type GO to get to start of main program
DBG>
```

The NOTATMAIN message indicates that execution is suspended before the start of the main program, so that you can execute initialization code under debugger control. Typing the GO command places you at the start of the main program. At that point, type the GO command again to start program execution. Execution continues until it is forced to pause or stop (for example, if the program prompts you for input, or an error occurs).

The following message indicates that your program has completed successfully:

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

To interrupt a debugging session and return to the DCL level, press Ctrl/y. This is useful if, for example, your program loops or you want to interrupt a debugger command that is still in progress.

To resume the debugging session after a Ctrl/y interruption, type either the CONTINUE or the DEBUG command at the DCL level. Use the CONTINUE command to return to the point at which you interrupted the debugging session. If you interrupted the session because of an infinite loop, use the DEBUG command instead. The DEBUG command returns you to the debugger prompt so that you can type another command. For example:

```
DBG> GO
   .
   .
   .
Ctrl/y
Interrupt

$ DEBUG
DBG>
```

To end a debugging session, type the EXIT command or press Ctrl/z:

```
DBG> EXIT
$
```

3.3.3 Issuing Debugger Commands

You can issue debugger commands anytime you see the debugger prompt (DBG>). Type the command at the keyboard and press the Return key. You can issue several commands on a line by separating the command strings with semicolons (;). As with DCL commands, you can continue a command string on a new line by ending the previous line with a hyphen (-).

Alternatively, you can use the numeric keypad to issue certain commands. Figure 3–1 identifies the predefined key functions. You can also redefine key functions with the DEFINE/KEY command.

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

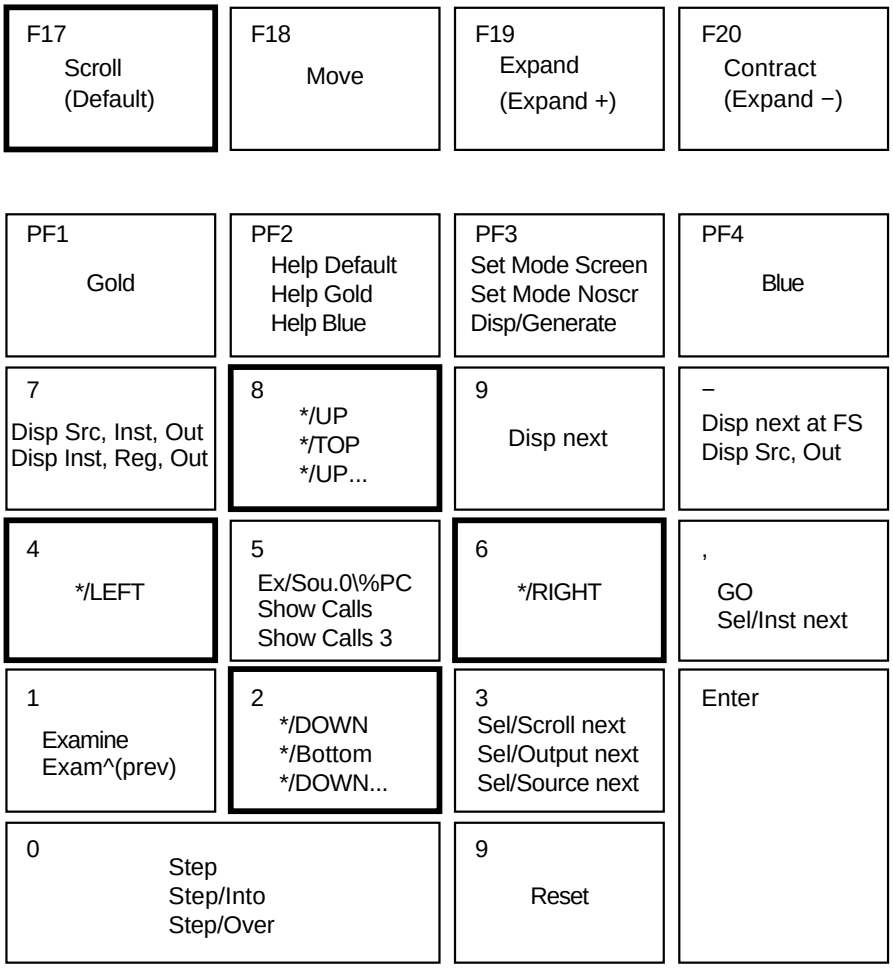
In Figure 3–1, the default, GOLD, and BLUE functions are listed within each key's outline, from top to bottom, respectively. For example, pressing keypad key 0 issues the STEP command; pressing key PF1 and then key 0 issues the STEP/INTO command; pressing key PF4 and then key 0 issues the STEP/OVER command.

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

3.3.4 Viewing Your Source Code

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

Figure 3–1 Debugger Keypad Key Functions



On a LK201 Keyboard:

Press	Keys 2, 4, 6, 8
F17	+ * = Scroll
F18	+ * = Move
F19	+ * = Expand
F20	+ * = Contract

On a VT−100 Keyboard:

Type	Keys 2, 4, 6, 8
SET KEY/STATE=DEFAULT	+ * = Scroll
SET KEY/STATE=MOVE	+ * = Move
SET KEY/STATE=EXPAND	+ * = Expand
SET KEY/STATE=CONTRACT	+ * = Contract

NU−2509A−RA

3.3.4.1 Noscreen Mode

Noscreen mode is the default, line-oriented mode of displaying input and output. To invoke noscreen mode from screen mode, press the keypad key sequence GOLD-PF3. See the sample debugging session in Section 3.4 for a demonstration of noscreen mode.

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

```
DBG> TYPE 3
3:    I = 7;
DBG>
```

The display of source lines is independent of program execution. To display source code from a module other than the one whose code is currently executing, use the TYPE command with a path name to specify the module. For example, the following command displays lines 16 through 21 of module TEST:

```
DBG> TYPE TEST\16:21
```

3.3.4.2 Screen Mode

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

```
- SRC: module MAIN -scroll-source-----------------------
     1: MAIN:  PROCEDURE OPTIONS (MAIN);
     2: DECLARE (I,J,K) FIXED BINARY;
     3:   I = 7;
->   4:   J = 4;
     5:   K = I + J;
     6: END;




- OUT -output-------------------------------------------
stepped to MAIN\%LINE 4
MAIN\I: 7
MAIN\J: 50331649


- PROMPT -error-program-prompt--------------------------
DBG> STEP 2
DBG> EXAMINE I,J
DBG>
```

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

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

The PROMPT display, at the bottom of the screen, shows the debugger prompt (DBG>), your input, debugger diagnostic messages, and program output. In the example, the two debugger commands that have been issued (STEP 2 and EXAMINE I,J) are displayed.

(The unpredictable value reported by the debugger for J indicates that line 4 has not been executed yet; line 4 will subsequently assign the value 4 to J.)

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

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

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

Source lines may not be available for the following reasons:

- The PC value is within a system routine or a shareable image routine for which no source code is available.
- The PC value is within a routine that was compiled without the /DEBUG compiler command qualifier (or with /NODEBUG).
- The PC value is within a routine whose module is not set (module setting is explained in Section 3.3.7.1).
- The source file was moved to a different directory after it was compiled (the location of source files is embedded in the object modules).

3.3.5 Controlling and Monitoring Program Execution

This section discusses the following topics:

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

3.3.5.1 Starting and Resuming Program Execution

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

The GO Command

The GO command starts program execution, which continues until forced to stop. The GO command is used most often in conjunction with breakpoints, tracepoints, and watchpoints (described in Sections 3.3.5.3, 3.3.5.4, and 3.3.5.5). If you set a breakpoint in the path of execution and then issue the GO command, execution is suspended at that breakpoint. If you set a tracepoint, the path of execution through that tracepoint is monitored. If you set a watchpoint, execution is suspended when the value of the watched variable changes.

You can also use the GO command to test for an exception condition or an infinite loop. If an exception condition that is not handled by your program occurs, the debugger takes control and displays the DBG> prompt so that you can issue commands. If you are using screen mode, the pointer in the source display indicates where execution stopped. You can use the SHOW CALLS command (explained in Section 3.3.5.2) to identify the currently active routine calls (the call stack).

If an infinite loop occurs, the program does not terminate, so the debugger prompt does not reappear. To obtain the prompt, interrupt the program by pressing Ctrl /y and then issue the DCL command DEBUG. You can then look at the source display and invoke a SHOW CALLS display to obtain the current PC value.

The STEP Command

The STEP command allows you to execute a specified number of source lines or instructions, or to execute the program to the next instruction of a particular kind, for example, to the next CALL instruction.

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

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

The PC value is now at the first machine code instruction for line 27 of the module TEST; line 27 is in COUNT, a routine within the module TEST. TEST\COUNT\%LINE 27 is a *path name*. The debugger uses path names to refer to symbols. (You do not need to use a path name in referring to a symbol, however, unless the symbol is not unique. If the symbol is not unique, the debugger issues an error message. See Section 3.3.7.2 for more information on resolving multiply defined symbols.)

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

```
DBG> STEP 3
```

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

Also, if a line contains more than one statement, the debugger executes all the statements on that line as part of the single step.

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

3.3.5.2 Determining the Current Value of the Program Counter

The SHOW CALLS command lets you determine the current value of the program counter (PC) (for example, after returning to the debugger following a Ctrl/y interruption).

The SHOW CALLS command displays a traceback that lists the sequence of calls leading to the currently executing routine. For each routine (beginning with the currently executing routine), the debugger displays the following information:

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

For example:

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

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

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

3.3.5.3 Suspending Program Execution

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

In the following example, the SET BREAK command sets a breakpoint on the procedure COUNT. The GO command then starts execution. When the procedure COUNT is encountered, execution is suspended. The debugger reports that the breakpoint at COUNT has been reached (`break at . . .`), displays the source line (54) where execution is suspended, and prompts you for another command. At this breakpoint, you could step through the procedure COUNT, using the STEP command, and use the EXAMINE command (discussed in Section 3.3.6.1) to check on the current values of X and Y.

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

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

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

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

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

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

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

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

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

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

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

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

If any portion of your program was written in Ada, two breakpoints that are associated with Ada tasking exception events are automatically established when you invoke the debugger. When you issue a SHOW BREAK command under these conditions, the following breakpoints are displayed:

```
DBG> SHOW BREAK
Breakpoint on ADA event "DEPENDENTS_EXCEPTION" for any value
Breakpoint on ADA event "EXCEPTION_TERMINATED" for any value
```

These breakpoints are equivalent to issuing the following commands:

```
DBG> SET BREAK/EVENT=DEPENDENTS_EXCEPTION
DBG> SET BREAK/EVENT=EXCEPTION_TERMINATED
```

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

3.3.5.4 Tracing Program Execution

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

As with breakpoints, every time a tracepoint is reached, the debugger issues a message and displays the source line. However, at tracepoints, the program continues executing, and the debugger prompt is not displayed. For example:

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

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

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

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

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

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

3.3.5.5 Monitoring Changes in Variables

The SET WATCH command lets you set *watchpoints* that will be monitored continuously as your program executes. With high-level languages, you typically set watchpoints on variables that have been declared in your program. In addition, you can set watchpoints on arbitrary program locations. If the program modifies the value of a watched variable, the debugger suspends execution and displays the old and new values.

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

```
DBG> SET WATCH TOTAL
```

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

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

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

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

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

Note that this general technique for setting watchpoints always applies to *static* variables. A static variable is associated with the same virtual memory location throughout program execution. In PL/I for OpenVMS VAX, only the following kinds of variables are statically allocated:

STATIC
EXTERNAL
GLOBALDEF
GLOBALREF

A variable that is allocated on the stack or in a register (a *nonstatic* variable) exists only when its defining routine is active (on the call stack). In PL/I for OpenVMS VAX nonstatic variables include the following:

AUTOMATIC
BASED
CONTROLLED
DEFINED
PARAMETER

If you try to set a watchpoint on a nonstatic variable when its defining routine is not active, the debugger issues a warning like the following:

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'Y' is not active
```

A convenient technique for setting a watchpoint on a nonstatic variable is to set a breakpoint on the defining routine, also specifying a DO clause to set the watchpoint whenever execution reaches the breakpoint. In the following example, a watchpoint is set on the nonstatic variable Y in routine COUNTER:

```
DBG> SET BREAK COUNTER DO (SET WATCH Y)
DBG> GO
   .
   .
   .
break at routine MOD4\COUNTER
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG> SHOW WATCH
watchpoint of MOD4\COUNTER\Y [tracing every instruction]
DBG>
```

The debugger monitors nonstatic watchpoints by tracing every instruction. Because this slows execution speed compared to monitoring static watchpoints, the debugger informs you when it is monitoring nonstatic watchpoints.

When execution eventually returns to the calling routine, the nonstatic variable is no longer active, so the debugger automatically cancels the watchpoint and issues a message to that effect.

3.3.6 Examining and Manipulating Data

This section explains how to use the EXAMINE, DEPOSIT, and EVALUATE commands to display and modify the contents of variables and to evaluate expressions. It also notes restrictions on the use of these commands with PL/I for OpenVMS VAX programs.

Note that, before you can examine or deposit into a nonstatic variable (as defined in the previous section), its defining routine must be active (on the call stack).

3.3.6.1 Displaying the Values of Variables

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

EXAMINE variable-name

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

Examine a string variable:

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

Examine three integer variables:

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

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

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

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

```
DBG> EXAMINE CHAR_ARRAY(4)
PROG2\CHAR_ARRAY(4): "m"
DBG>
```

Examine the value of a variable declared as FIXED DECIMAL (10,5):

```
DBG> EXAMINE X
PROG4\X:   540.02700
DBG>
```

Examine the value of a structure variable:

```
DBG> EXAMINE PART
MAIN_PROG\INVENTORY_PROG.PART
    ITEM:      "WF-1247"
    PRICE:       49.95
    IN_STOCK:    24
DBG>
```

Examine the value of a pictured variable (note that the debugger displays the value in quotation marks):

```
DBG> EXAMINE Q
MAINP\Q:    "666.3330"
DBG>
```

Examine the value of a pointer (which is the virtual address of the variable it accesses) and display the value in hexadecimal radix instead of decimal (the default):

```
DBG> EXAMINE/HEXADECIMAL P
PROG4\SAMPLE.P: 0000B2A4
DBG>
```

Examine the value of a variable with the BASED attribute; in this case, the variable X has been declared as BASED(PTR), with PTR its pointer:

```
DBG> EXAMINE X
PROG5\X:    "A"
DBG>
```

Examine the value of a variable X declared as BASED with a variable PTR declared as POINTER; here, PTR is associated with X by the following line of PL/I code (instead of X having been declared as BASED(PTR) as in the preceding example):

```
ALLOCATE X SET (PTR);
```

In this case, you examine the value of X as follows:

```
DBG> EXAMINE PTR->X
PROG6\PTR->X:    "A"
DBG>
```

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

See Section 3.3.6.3 for a comparison of the EXAMINE and EVALUATE commands.

3.3.6.2 Changing the Values of Variables

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

```
DEPOSIT variable-name = value
```

The DEPOSIT command is like an assignment statement in PL/I for OpenVMS VAX.

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

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

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

Deposit an integer expression:

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

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

```
DBG> DEPOSIT C_ARRAY(12) = 'K'
```

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

3.3.6.3 Evaluating Expressions

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

```
EVALUATE language-expression
```

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

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

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

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

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

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

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

3.3.6.4 Notes on Debugger Support for PL/I

In general, the debugger supports the data types and operators of PL/I and of the other debugger-supported languages. However, there are certain language-specific limitations or other differences. (For information on the supported data types and operators of any of the languages, type the HELP LANGUAGE command at the DBG> prompt.)

You cannot use the DEPOSIT command with entry or label variables or formats, or with entire arrays or structures. You cannot use the EXAMINE command with entry or label variables or formats; use the EVALUATE/ADDRESS command instead.

You cannot use the EXAMINE command to determine the values or attributes of global literals (such as GLOBALDEF VALUE literals) because they are static expressions. Use the EVALUATE command instead.

You cannot use the EXAMINE, EVALUATE, and DEPOSIT commands with compile-time variables and procedures. You can, however, use EVALUATE and DEPOSIT (but not EXAMINE) with a compile-time constant, as long as the constant is the source and not the destination.

Note that an uninitialized automatic variable does not have valid contents until after a value has been assigned to it. If you examine it before that point, the value displayed is unpredictable.

You can deposit a value into a pointer variable either by depositing another pointer's value into it, thus making symbolic reference to both pointers, or by depositing a virtual address into it. (You can find out the virtual address of a variable by using the EVALUATE/ADDRESS command, and then deposit that address into the pointer.) When you examine a pointer, the debugger displays its value in the form of the virtual address of the variable that the pointer points to.

Because the default representation of decimal constants in PL/I is packed decimal rather than fixed binary, debugger commands such as EVALUATE/HEXADECIMAL 53 display the number's packed decimal representation. You can cause conversion to take place, however, either by specifying a fixed binary variable in the expression, or by using a constant in a radix other than decimal (because nondecimal radix constants are assumed to be fixed binary), as in the command:

```
EVALUATE/HEXADECIMAL 53 + %HEX 0.
```

3.3.7 Controlling Symbol References

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

- Module setting
- Multiply defined symbols

3.3.7.1 Module Setting

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

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

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

If you try to reference a symbol in a module that has not been set, the debugger issues a warning. For example:

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

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

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

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

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

3.3.7.2 Resolving Multiply Defined Symbols

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

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

If the debugger cannot resolve a symbol ambiguity, it issues a warning. For example:

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

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

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

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

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

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

3.4 Sample Debugging Session

This section shows a sample debugging session with a PL/I program, SUM, which contains a logic error. Line numbers have been added to facilitate the discussion.

```
 1            SUM: PROCEDURE OPTIONS(MAIN);
 2    1            DECLARE (I,HIGHEST,TOTAL) FIXED;
 3    1               TOTAL=0;
 4    1               HIGHEST=1;
 5    1               DO WHILE (HIGHEST>0);
 6    2                   GET LIST (HIGHEST) OPTIONS (PROMPT
 7    2                    ('Type a number greater than 0, or 0 to quit: '));
 8    2                   IF HIGHEST <=0
 9    2                    THEN
10    2                       STOP;
11    2                   DO I=1 TO HIGHEST;
12    3                       TOTAL=TOTAL+I;
13    3                       END;
14    2                   PUT SKIP EDIT
15    2                    ('The sum of integers from 1 through',
16    2                    HIGHEST,' is',TOTAL)
17    2                    (A,F(10),A,F(10));
18    2                   PUT SKIP;
19    2                   END;  /* DO WHILE */
20    1        END SUM;
```

This program prompts for a number and prints the sum of the integers from 1 through the number entered. The problem in the program occurs because the variable TOTAL is not reinitialized when a new number is entered; the statement assigning the value 0 to TOTAL occurs before the loop instead of within it.

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

```
$ PLI SUM
$ LINK SUM
$ RUN SUM
Type a number greater than 0, or 0 to quit: 5

The sum of integers from 1 through         5 is        15
Enter a number: 4
The sum of integers from 1 through         4 is        25
Type a number greater than 0, or 0 to quit: 0
$
```

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

To debug the program, you must compile and link with the debugger. (If you want a listing with line numbers to refer to during the debugging session, include the /LIST qualifier with the PLI command, and then print the listing file that results; you need not specify the file type because the PRINT command searches for the LIS file type by default.) For example:

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

You are now ready to begin a debugging session. The terminal session is keyed to the numbered notes that follow.

```
$ RUN SUM

                    VAX DEBUG Version <VMS_VERSION>
```

```
%DEBUG-I-INITIAL, language is PLI, module set to 'SUM'  1
DBG> SET BREAK %LINE 7   2
DBG> GO   3
break at SUM\%LINE 7   4
     7:              ('Type a number greater than 0, or 0 to quit: '));
DBG> EXAMINE TOTAL
SUM\TOTAL:      0   5
DBG> GO
Type a number greater than 0, or 0 to quit: 5

The sum of integers from 1 through         5 is        15   6

break at SUM\%LINE 7
     7:              ('Type a number greater than 0, or 0 to quit: '));   7
DBG> EXAMINE TOTAL
SUM\TOTAL:     15   8
DBG> DEPOSIT TOTAL=0   9
DBG> GO
Type a number greater than 0, or 0 to quit: 4
The sum of integers from 1 through         4 is        10   10

break at SUM\%LINE 7
     7:              ('Type a number greater than 0, or 0 to quit: '));
DBG> GO
Type a number greater than 0, or 0 to quit: 0 11
%DEBUG-I-EXITSTATUS, is '%SYSTEM-S-NORMAL, normal successful completion'
DBG> EXIT   12
$
```

1 When you issue the RUN command, the debugger displays an informational message and the DBG> prompt. You are now in the default noscreen mode. The lines of source code are displayed as they are executed, by default.

2 You decide that the problem may lie with the initialization of the variable TOTAL. You can test this hypothesis by examining the value of TOTAL each time you enter a new number. To stop the program at the point at which you can do this, you set a breakpoint at line 7.

3 The GO command starts program execution.

4 When line 7 is reached, the debugger interrupts program execution and displays the source line at which the breakpoint was set.

5 You use the EXAMINE command to determine the current value of the variable TOTAL. Its value is 0, as expected at this point.

6 The GO command resumes program execution. The program now prompts you for a number. You type 5. The program's response is correct.

7 The debugger again reaches the breakpoint at line 7 and displays the source line.

8 You examine the variable TOTAL with the EXAMINE command. Its value is 15, not 0 as it should be. This indicates that the assignment statement that initializes TOTAL is misplaced.

9 The DEPOSIT command replaces the contents of TOTAL with 0, thus allowing the program to return a correct result the next time through the loop.

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

11 When you enter a 0 in response to the prompt, the program exits, causing the debugger to display a message that indicates the termination status.

12 The EXIT command terminates the debugging session.

You can now correct the program so that it reinitializes the variable TOTAL correctly.

3.5 Debugger Command Summary

This section lists all of the debugger commands and any related DCL commands in functional groupings, along with brief descriptions.

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

HELP debugger command

3.5.1 Starting and Terminating a Debugging Session

($) RUN[1]	Invokes the debugger if LINK/DEBUG was used
($) RUN/[NO]DEBUG[1]	Controls whether the debugger is invoked when the program is executed
Ctrl/z or EXIT	Ends a debugging session, executing all exit handlers
QUIT	Ends a debugging session without executing any exit handlers declared in the program
Ctrl/y	Interrupts a debugging session and returns you to the DCL level
Ctrl/c	Has the same effect as Ctrl/y, unless the program has a Ctrl/c service routine
($) CONTINUE[1]	Resumes a debugging session after a Ctrl/y interruption
($) DEBUG[1]	Resumes a debugging session after a Ctrl/y interruption but returns you to the debugger prompt
ATTACH	Passes control of your terminal from the current process to another process (similar to the DCL command ATTACH)
SPAWN	Creates a subprocess; lets you issue DCL commands without interrupting your debugging context (similar to the DCL command SPAWN)

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

3.5.2 Controlling and Monitoring Program Execution

GO	Starts or resumes program execution
STEP	Executes the program up to the next line, instruction, or specified instruction
{ SET / SHOW } STEP	Establishes or displays the default qualifiers for the STEP command
{ SET / SHOW / CANCEL } BREAK	Sets, displays, or cancels breakpoints
{ SET / SHOW / CANCEL } TRACE	Sets, displays, or cancels tracepoints

{ SET | SHOW | CANCEL } WATCH	Sets, displays, or cancels watchpoints
{ SET | CANCEL } EXCEPTION BREAK	Sets or cancels exception breakpoints
SHOW CALLS	Identifies the currently active routine calls
SHOW STACK	Gives additional information about the currently active routine calls
CALL	Calls a routine

3.5.3 Examining and Manipulating Data

EXAMINE	Displays the value of a variable or the contents of a program location
DEPOSIT	Changes the value of a variable or the contents of a program location
EVALUATE	Evaluates a language or address expression

3.5.4 Controlling Type Selection and Symbolization

{ SET | SHOW | CANCEL } RADIX	Establishes the radix for data entry and display, displays the radix, or restores the radix
{ SET | SHOW | CANCEL } TYPE	Establishes the type to be associated with untyped program locations, displays the type, or restores the type
SET MODE [NO]G_FLOAT	Controls whether double-precision floating-point constants are interpreted as G_FLOAT or D_FLOAT
SET MODE [NO]LINE	Controls whether code locations are displayed in terms of line numbers or routine-name + byte offset
SET MODE [NO]SYMBOLIC	Controls whether code locations are displayed symbolically or in terms of numeric addresses
SYMBOLIZE	Converts a virtual address to a symbolic address

3.5.5 Controlling Symbol Lookup

SHOW SYMBOL	Displays symbols in your program
{ SET | SHOW | CANCEL } MODULE	Sets a module by loading its symbol records into the debugger's symbol table, identifies a set module, or cancels a set module
{ SET | SHOW | CANCEL } IMAGE	Sets a shareable image by loading data structures into the debugger's symbol table, identifies a set image, or cancels a set image

SET MODE [NO]DYNAMIC	Controls whether modules and shareable images are set automatically when the debugger interrupts execution
{ SET / SHOW / CANCEL } SCOPE	Establishes, displays, or restores the scope for symbol lookup

3.5.6 Displaying Source Code

TYPE	Displays lines of source code
EXAMINE/SOURCE	Displays the source code at the location specified by the address expression
{ SET / SHOW / CANCEL } SOURCE	Creates, displays, or cancels a source directory search list
SEARCH	Searches the source code for the specified string
{ SET / SHOW } SEARCH	Establishes or displays the default qualifiers for the SEARCH command
{ SET / SHOW } MAX_SOURCE_FILES	Establishes or displays the maximum number of source files that may be kept open at one time
{ SET / SHOW } MARGINS	Establishes or displays the left and right margin settings for displaying source code

3.5.7 Using Screen Mode

SET MODE [NO]SCREEN	Enables or disables screen mode
SET MODE [NO]SCROLL	Controls whether an output display is updated line by line or once per command
DISPLAY	Modifies an existing display
{ SET / SHOW / CANCEL } DISPLAY	Creates, identifies, or deletes a display
{ SET / SHOW / CANCEL } WINDOW	Creates, identifies, or deletes a window definition
SELECT	Selects a display for a display attribute
SHOW SELECT	Identifies the displays selected for each of the display attributes
SCROLL	Scrolls a display
SAVE	Saves the current contents of a display and writes it to another display
EXTRACT	Saves a display or the current screen state and writes it to a file
EXPAND	Expands or contracts a display

MOVE	Moves a display across the screen
{ SET / SHOW } TERMINAL	Establishes or displays the height and width of the screen
Ctrl/w or DISPLAY/REFRESH	Refreshes the screen

3.5.8 Editing Source Code

EDIT	Invokes an editor during a debugging session
{ SET / SHOW } EDITOR	Establishes or identifies the editor invoked by the EDIT command

3.5.9 Defining Symbols

DEFINE	Defines a symbol as an address, command, or value
DELETE or UNDEFINE	Deletes symbol definitions
{ SET / SHOW } DEFINE	Establishes or displays the default qualifier for the DEFINE command
SHOW SYMBOL/DEFINED	Identifies symbols that have been defined

3.5.10 Using Keypad Mode

SET MODE [NO]KEYPAD	Enables or disables keypad mode
DEFINE/KEY	Creates key definitions
DELETE/KEY or UNDEFINE/KEY	Deletes key definitions
SET KEY	Establishes the key definition state
SHOW KEY	Displays key definitions

3.5.11 Using Command Procedures and Log Files

DECLARE	Defines parameters to be passed to command procedures
{ SET / SHOW } LOG	Specifies or identifies the debugger log file
SET OUTPUT [NO]LOG	Controls whether a debugging session is logged
SET OUTPUT [NO]SCREEN_LOG	Controls whether, in screen mode, the screen contents are logged as the screen is updated
SET OUTPUT [NO]VERIFY	Controls whether debugger commands are displayed as a command procedure is executed
SHOW OUTPUT	Displays the current output options established by the SET OUTPUT command
{ SET / SHOW } ATSIGN	Establishes or displays the default file specification that the debugger uses to search for command procedures
@file-spec	Executes a command procedure

3.5.12 Using Control Structures

IF	Executes a list of commands conditionally
FOR	Executes a list of commands repetitively
REPEAT	Executes a list of commands repetitively
WHILE	Executes a list of commands conditionally
EXITLOOP	Exits an enclosing WHILE, REPEAT, or FOR loop

3.5.13 Additional Commands

SET PROMPT	Specifies the debugger prompt
SET OUTPUT [NO]TERMINAL	Controls whether debugger output is displayed or suppressed, except for diagnostic messages
{ SET / SHOW } LANGUAGE	Establishes or displays the current language
{ SET / SHOW } EVENT_FACILITY	Establishes or identifies the current run-time facility for language-specific events
SHOW EXIT_HANDLERS	Identifies the exit handlers declared in the program
{ SET / SHOW } TASK	Modifies the tasking environment or displays task information
{ DISABLE / ENABLE / SHOW } AST	Disables the delivery of ASTs in the program, enables the delivery of ASTs, or identifies whether delivery is enabled or disabled

4
The File System

This chapter discusses the use of files for input and output (I/O) in PL/I for OpenVMS VAX and PL/I for OpenVMS AXP and describes the aspects of the OpenVMS operating system that relate to PL/I I/O. The chapter includes the following topics:

- Declaring, opening, and closing files
- The relationship between PL/I I/O statements and OpenVMS I/O procedures
- OpenVMS file-naming and file-defining conventions, including a description of logical names and process permanent files
- File system error handling at run time

Chapters 5, 6, 7, 8, 9 give additional information on the file system, covering stream and record I/O, options (ENVIRONMENT and I/O statement options), and built-in subroutines for file handling.

4.1 File Control

File constants and variables provide your program with access to files. Your program first declares a file constant or variable, and then associates the constant or variable with a file when it opens the file.

A file declaration specifies an identifier and the FILE attribute, and optionally specifies one or more file description attributes describing the type of I/O operation that will be used to process the file. Subsequent I/O statements denote the file by a FILE option.

The OPEN statement explicitly opens a PL/I file with a specified set of attributes that describe the file and the method for accessing it. A file can also be opened implicitly by a READ, WRITE, REWRITE, DELETE, PUT, or GET statement issued for a file that is not open, or by a built-in subroutine referring to a file that is not open.

When PL/I opens a file, the initial positioning depends on the type of file (record or stream), the access mode, and certain ENVIRONMENT options. File positioning for stream files and record files is described in Chapters 5 and 6, respectively.

The CLOSE statement dissociates a PL/I file from the physical file with which it was associated when it was opened. Some ENVIRONMENT options are valid in the CLOSE statement. ENVIRONMENT options are summarized in Chapter 7.

Declaring, opening, and closing files are described in more detail in the *PL/I for OpenVMS Systems Reference Manual*.

4.2 Using the OpenVMS File System for I/O

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

In the OpenVMS system, I/O is performed by the following services:

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

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

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

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

4.2.1 PL/I Files and OpenVMS File Specifications

In a PL/I program, all I/O operations are performed on a file, using the name of a file constant or file variable. When the file is opened, PL/I associates the name of the file constant with a specific device or file on the computer system.

When a file variable is specified in an OPEN statement or in an I/O statement, the name used is that of the file constant with which the variable is currently associated. For example:

```
DECLARE F FILE,
        G FILE VARIABLE;

        G = F;
        OPEN FILE(G);
```

In this example, F is a file constant and G is a file variable assigned the value of F. In the OPEN statement, PL/I uses the name F to associate the PL/I file with an OpenVMS file. The default file would be F.DAT.

The following sections describe in more detail how PL/I for OpenVMS VAX and PL/I for OpenVMS AXP associate a file constant with a device or file.

4.2.2 Using the TITLE Option

When you specify the TITLE option on an OPEN statement, you can include all or part of an OpenVMS file specification to indicate the file or device to be associated with the PL/I file. The following examples illustrate the use of the TITLE option.

```
OPEN FILE (OUTFILE)
     TITLE('DB1:[PAYROLL.DAT]JANUARY.LOG;2');
```

This file specification completely defines a file on the local OpenVMS system.

```
OPEN FILE (PRINTFILE) PRINT TITLE('LPC0:SAMPLE.DAT');
```

This output file will be directed to LPC0:, the system printer device. The listing file will have the title SAMPLE.DAT on its burst page. OpenVMS spools low-speed I/O devices such as printers by accumulating data for the device in a file, and then queuing the file for processing when it is closed.

```
NAME = 'TEST'||COUNT;
   .
   .
   .
OPEN FILE(NEWFILE) OUTPUT TITLE(NAME);
```

The specification of this file is determined by the value of COUNT. For example, if COUNT is 5 when this OPEN statement executes, the file created is TEST5.DAT.

When no TITLE option is specified, PL/I supplies a default value for the file's title. The default title is the name of the file constant associated with the PL/I file. Whenever a title does not completely specify a file, PL/I for OpenVMS VAX or PL/I for OpenVMS AXP takes the following steps, in order:

1. It performs logical name translation. If there is a colon (:) present in the TITLE option, the file system attempts to find an equivalence name for the portion of the file specification on the left of the colon. If there are no punctuation marks in the TITLE option, the file system attempts to find an equivalence name for the entire specification.
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 applies system defaults to complete the file specification.

If the file specification that is finally achieved is invalid or represents an illegal device or file (for example, an input file cannot be found), the UNDEFINEDFILE condition is signaled.

4.2.3 Using Logical Names

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

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

Before running the program, you might associate an OpenVMS file with the identifier INFILE:

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

The DEFINE command assigns the PL/I file INFILE the OpenVMS file specification DB1:[TEMP]A.DAT. In OpenVMS 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 issue the following command:

```
$ 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 can 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 translates the name INFILE and uses the tape MTA1: as the input device.

Logical Names in TITLE Values

The value specified in a TITLE option can represent a logical name. For example:

```
OPEN FILE(INFILE) TITLE ('NEWFILE');
```

This file might be defined as follows:

```
$ DEFINE NEWFILE LARGO.TXT
```

When the OPEN statement opens the file INFILE, the file's title (NEWFILE) is translated to LARGO.TXT. If NEWFILE is not defined when the OPEN statement is executed, the OPEN statement opens the file NEWFILE.DAT.

If instead of containing a colon a file specification in the TITLE option contains any of the punctuation marks that are used in OpenVMS file specifications, the file system does not translate that portion of the specification. For example:

```
TITLE ('STATES.')
```

In this example, the file system assumes that 'STATES.' is the specification of a file name and a file type (the presence of the period indicates a null file type). It does not perform any translation.

When you enter a complete file specification for the TITLE option and you do not want the file system to attempt logical name translation, you can precede the file specification with an underscore (_). For example:

```
TITLE ('_DBB1:[APPLIC.FILES]MASTER.SRC')
```

The system does not perform translation if an underscore precedes either a device name or a file name that is specified with no punctuation.

Process Permanent Logical Names

The system provides every user and every batch job with a default set of process logical name assignments, which are listed in Table 4–1. 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.

Table 4–1 Default Process Logical Names

Logical Name	Default Equivalence Name
SYS$INPUT	Input stream. For an interactive user, this is the terminal or a command procedure file; for a batch job, the input command file.
SYS$OUTPUT	Output stream. For an interactive user, this is the terminal; for a batch job, the batch job log file.
SYS$ERROR	Error stream. Unless overridden by the user, it is the same as SYS$OUTPUT.
SYS$DISK	Default device.
SYS$COMMAND	Default command stream. For an interactive user, this is the terminal; for a batch job, the batch job input command file.

The default files associated with the GET and PUT statements, SYSIN and SYSPRINT, are defined by PL/I as follows:

Statement	PL/I File	Default Title
GET	SYSIN	SYS$INPUT
PUT	SYSPRINT	SYS$OUTPUT

Thus, when your program executes a GET statement that does not specify the FILE option, and if SYSIN was not explicitly opened with a title, the run-time system and the file system perform the following translations:

1. PL/I attempts to translate the logical name SYSIN. If no logical name assignment exists for it, PL/I replaces the name SYSIN with the name SYS$INPUT.
2. The system translates the logical name SYS$INPUT. The resulting file specification is your current input device.

A similar set of associations occurs when a program executes a PUT statement without the FILE option: the resulting output is written to the current output file, SYS$OUTPUT.

4.2.4 Using the DEFAULT_FILE_NAME Option

Use the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute to specify default values for a file specification when a file is opened.

For example:

```
OPEN FILE (REPORT) RECORD OUTPUT
          ENVIRONMENT (
                   DEFAULT_FILE_NAME('.LIS'));
```

The default file type LIS will be applied in the following cases:

- No logical name assignment exists for the name REPORT when the program containing this statement is executed. In this case, the file will be named REPORT.LIS, and it will be cataloged in the current default directory.
- 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.

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP use the punctuation in the DEFAULT_FILE_NAME option to determine which portion of the file specification is specified. Thus, the period (.) in the preceding 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, and no file type is specified, PL/I supplies the default file type DAT.

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 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 follows:

```
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 statement opens the file RONDO::DBB2:[MALCOLM]ALLEGRO.TXT.

Another OPEN statement for the file can specify a different TITLE option, for example, TITLE('ANDANTE'), to open a different file.

4.2.5 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

The following examples enumerate the steps in completing a file specification. All examples assume that the following logical name assignments are in effect:

```
TAPEFILE      = MTA0:
STATE_NAME    = NEBRASKA
STATES        = DMA2:[BACKUP]
```

They also assume the following current default device and directory:

```
DBB1:[MALCOLM]
```

In the following example, the value in the TITLE option represents the complete file specification:

```
DECLARE STATES FILE RECORD OUTPUT;
OPEN FILE(STATES) TITLE ('_DMA2:[STATE.DATA]NEVADA.DAT;2');
```

Thus, the final specification is: _DMA2:[STATE.DATA]NEVADA.DAT;2

The following example uses several steps to obtain the file specification:

```
DCL STATES FILE INPUT ENVIRONMENT(
                    DEFAULT_FILE_NAME ('[STATE.FILES].DAT'));
OPEN FILE(STATES) TITLE ('MISSOURI');
```

The steps are:

1. Obtain the value specified in the TITLE option, MISSOURI.
2. Since there is no logical name assignment for MISSOURI, use MISSOURI as the file name.
3. Apply the value in the DEFAULT_FILE_NAME option, which includes a default directory, [STATE.FILES], and a default file type, DAT.
4. Apply system defaults for device and version number (for an input file).
5. The final specification is: DBB1:[STATE.FILES]MISSOURI.DAT;n where n is the highest existing version of the file.

The following example includes the translation of a logical name:

```
DECLARE STATES FILE RECORD OUTPUT;
OPEN FILE(STATES) TITLE ('STATE_NAME');
```

The steps are:

1. Obtain the value from the TITLE option, STATE_NAME.
2. Translate the logical name STATE_NAME to obtain the equivalence, NEBRASKA.
3. Apply default device, directory, file type, and version number (for an output file).
4. The final specification is: DBB1:[MALCOLM]NEBRASKA.DAT;n where n is 1 higher than the number of any existing version of the file.

In the following example, no title is specified and the default title is applied:

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

The steps are:

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.
4. The final specification is: DMA2:[BACKUP].DAT;n where n is 1 higher than the number of any existing version of the file.

The following example specifies a tape file:

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

The steps are:

1. Apply the title TAPE1.FIL.
2. Translate the logical name TAPEFILE in the DEFAULT_FILE_NAME option to MTA0:, its equivalent.
3. Use the system default version number for tape files, 0. Tape files do not have directories.
4. The final specification is: MTA0:TAPE1.FIL;0

4.3 Error Handling

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP use the standard PL/I ON condition names to signal run-time errors that occur for file operations. The ON conditions and the circumstances under which they are signaled are as follows:

- 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.

To handle any of these conditions in a PL/I procedure, you can establish an ON-unit to receive control if the specified condition is signaled. For example:

```
ON UNDEFINEDFILE (INFILE) OPEN FILE (INFILE)
             TITLE ('SYS$INPUT');
```

The ON statement provides a default title for the file INFILE.

4.3.1 Values Returned by PL/I Built-In Functions for Error Handling

An ON-unit can be a generalized error-handling routine, written so that it responds to specific errors or prints an error message. Following are the PL/I built-in functions that provide meaningful information in an ON-unit to handle a file system error:

- ONCODE
- ONFILE
- ONKEY

Whenever an error is signaled, the built-in function ONCODE makes available the condition value associated with the specific error. When a PL/I program is executing under control of the OpenVMS operating system, the value returned by the ONCODE built-in function is a unique 32-bit condition value that indicates the reason for the error. This value is from the system, from RMS, or from the PL/I run-time system.

The built-in function ONFILE returns a character string giving the name of the file constant on which the error occurred.

If the file was being accessed by key, the ONKEY built-in function returns the key value that caused the error to be signaled.

4.3.2 Writing an Error Handler

You can write an ON-unit to detect and correct errors that occur during file operations. The following example shows an ON-unit that detects whether a record with a given key value was not found or whether an attempt was made to write a record whose key duplicates the value of an existing key.

```
ON KEY(STATE_FILE) BEGIN;
  %INCLUDE $RMSDEF;

/* Check for a record not found */
     IF ONCODE() = RMS$_RNF       /* if record not found */
     THEN DO;
        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. This code indicates that a key value is invalid; for example, it is an incorrect data type.

The symbolic names for RMS status codes are declared as %REPLACE constants in module $RMSDEF of PLI$STARLET.

4.3.3 Default Error Handling for the File System

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 condition handler is present and executed. It prints a PL/I run-time error message. If there is no default PL/I handler, the error signal is passed to the default condition handler established by the OpenVMS system, 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 message 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:[PROJECT]STDATA.DAT;'
-PLI-I-NOTKEYD, Not a KEYED file.
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine          line   relative PC  absolute PC
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 lacking the KEYED attribute.

5
Stream Input/Output

Stream I/O is one of the two general kinds of I/O performed by PL/I (the other is record I/O, described in Chapter 6). In stream I/O, more than one record or line can be processed by a single statement, and, conversely, multiple statements can process a single line or record. In record I/O, only one record of a file is processed by each READ or WRITE statement.

Stream input and output are performed by the statements GET and PUT, respectively. Both statements can perform either list-directed or edit-directed operations.

This chapter discusses the physical organization of stream files. See the *PL/I for OpenVMS Systems Reference Manual* for detailed information on the following topics:

- The physical organization of stream files
- The options and attributes applicable to stream I/O—the LINESIZE and PAGESIZE options and the PRINT and STREAM attributes
- The statements used for stream I/O—GET, PUT, and FORMAT
- The processing and positioning that take place during stream I/O operations
- The format items and methods of combining them into format specifications for edit-directed stream I/O operations

A file has the STREAM attribute if one of the following conditions is met:

- The file was declared explicitly with a DECLARE statement and the STREAM attribute.
- The file was declared explicitly with a DECLARE statement and with neither the STREAM nor the RECORD attribute (as the default is STREAM).
- 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 the *PL/I for OpenVMS Systems Reference Manual*.

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.

An existing stream file can be reopened and accessed by the READ and WRITE statements. If READ and WRITE open a file implicitly, the attributes RECORD INPUT SEQUENTIAL and RECORD OUTPUT SEQUENTIAL are implied, respectively. These attributes are compatible with, for example, an existing disk file created with the STREAM and OUTPUT attributes.

The most straightforward way to retrieve an entire line (record) from an existing stream file is with a simple READ statement. The following sample program uses the READ statement to retrieve entire lines from a stream file (TEST.STR) that was created by a previous program; each record is then output to a record file (TEST.REC). Note that both files are declared with the RECORD attribute to be compatible with the READ and WRITE statements. TEST.STR is assumed to have 80-character lines, and these input records are assigned to a varying-length character variable.

```
FILES: PROCEDURE OPTIONS(MAIN);

%REPLACE LENGTH BY 80;

DCL INFILE RECORD INPUT;
DCL OUTFILE RECORD OUTPUT;
DCL INSTRING CHAR(LENGTH) VARYING;
DCL EOF BIT(1) INITIAL('0'B);

      ON ENDFILE(INFILE) EOF='1'B;

      OPEN FILE(INFILE) TITLE('TEST.STR');
      OPEN FILE(OUTFILE) TITLE('TEST.REC');

      READ FILE(INFILE) INTO(INSTRING);
      DO WHILE(^EOF);
          WRITE FILE(OUTFILE) FROM(INSTRING);
          READ FILE(INFILE) INTO(INSTRING);
          END;

END FILES;
```

You can perform comparable operations with GET and PUT, working on existing files that were created with the RECORD attribute. Note, however, that the GET and PUT statements always access a file sequentially, regardless of the file's physical organization, and that all their input or output data is in ASCII characters. Some types of record files may also contain, in a single record, both ASCII program data and other ASCII data used to manipulate record files. If the file's records contain information other than program data, that information must be interpreted and processed by your program.

6
Record Input/Output

Record I/O is performed by the READ, WRITE, DELETE, and REWRITE statements; each statement processes an entire record. (In stream I/O, more than one line or record can be processed by a single statement.) In addition, some forms of record I/O allow you to access records in the file by record number or by a key field contained in the record.

This chapter includes the following topics:

- File organizations for record files—sequential, relative, and indexed sequential
- Access modes—sequential, random, and random/sequential, as well as access through block I/O and record identification
- Record formats—variable-length, fixed-length, and variable with fixed-length control
- Carriage control
- Sequential files
- Relative files
- Indexed sequential files

For detailed description of record positional information and the READ, WRITE, DELETE, and REWRITE statements, see the *PL/I for OpenVMS Systems Reference Manual*.

6.1 File Organizations

VAX Record Management Services (RMS) supports the following three file organizations for record files:

- 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 again. When a program subsequently accesses an existing file, the file's organization and attributes are known to the file system. Table 6–1 shows the attributes and access modes for record files.

Sequential Files

In a sequential file, records are ordered one after the other, in the order in which they are written. New records can be added only at the end of an existing file. Existing records can be updated or replaced only if the replacement record is exactly the same length as the original record. Sequential files are discussed in more detail in Section 6.5.

Relative Files

A relative file contains a specified number of fixed-length cells, numbered from 1 to n, where n is a user-specified maximum. Any record written to the file is written to a specific numbered cell; the cell number is also the relative record number of the record in the file.

Normally, the records in a relative file have an implied numeric data field, for example, an account number or employee number, which corresponds to the cell in which the record will be placed. When you read or write a record in a relatively organized file, you specify the record by its relative record number. Relative files are discussed in more detail in Section 6.6.

Indexed Sequential Files

In an indexed sequential file, each record has one or more data keys that define a sort order for the file. The file system maintains an index for each key in the file and uses these indexes to locate a record when a program specifies the key of interest.

The records can be accessed either in the sort order defined by a key, or individually by key specification. Indexed sequential files are discussed in more detail in Section 6.7.

Table 6–1 shows the attributes and access modes for record files.

Table 6–1 Attributes and Access Modes for Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
SEQUENTIAL OUTPUT	RECORD	Any output device or file except indexed	Records can be added to the end of the file with WRITE statements. Each WRITE statement adds a single record to the file.
SEQUENTIAL INPUT	RECORD	Any input device or file	Records in the file are read with READ statements. Each statement reads a single record.
SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk	READ statements read a file's records in order. PL/I maintains the current record, which is the record just read. This record can be replaced in a REWRITE statement.[1] In a relative or indexed sequential file, the current record can also be deleted with a DELETE statement. Each statement processes a single record.
DIRECT OUTPUT	KEYED RECORD	Relative, indexed, sequential disk[2]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record.

[1]For a file with sequential organization, the record being written must have the same length as the one that was read.

[2]The file must have fixed-length records.

(continued on next page)

Table 6–1 (Cont.) Attributes and Access Modes for Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
DIRECT INPUT	KEYED RECORD	Relative, indexed, sequential disk[2]	READ statements specify records to be read randomly by key. Each statement reads a single record.
DIRECT UPDATE	KEYED RECORD	Relative, indexed, sequential disk[2]	READ, WRITE and REWRITE statements specify records randomly by key. In a relative or indexed file, records can also be deleted by key.
KEYED SEQUENTIAL OUTPUT	RECORD	Relative, indexed, sequential disk[2]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record. This mode is identical to DIRECT OUTPUT.
KEYED SEQUENTIAL INPUT	RECORD	Relative, indexed, sequential disk[2]	READ statements access records in the file randomly by key or sequentially.
KEYED SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk[1]	Any record I/O operation is allowed except a WRITE statement that does not specify a key or a DELETE statement for a sequential disk file with fixed-length records.

[1]For a file with sequential organization, the record being written must have the same length as the one that was read.

[2]The file must have fixed-length records.

6.2 Access Modes

In standard PL/I, you can specify one of the following sets of attributes to define the way a program is going to access the records in a file:

- SEQUENTIAL indicates sequential access to a sequential or keyed file.
- DIRECT indicates random access to a keyed file.
- KEYED SEQUENTIAL indicates sequential and random access to a keyed file.

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, there are two additional ways to access a file. You specify an ENVIRONMENT option to indicate that you are going to process a file in either of these ways:

- BLOCK_IO indicates that the file will be accessed in terms of blocks, rather than records.
- RECORD_ID_ACCESS indicates that records can be read or written randomly with the file address of the record.

Although you cannot change the organization of the records in a file after you have created the file, you can select a different type of access mode for reading or processing the file. You can also access a file using more than one access mode; for example, you can read a record in an indexed file by specifying a key, and then read records sequentially beginning at that position in the file.

6.2.1 Sequential Access

You can use sequential access with any type of file organization. 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 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 involve only cells that contain data records.

When you access an indexed sequential file sequentially, you can 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.

6.2.2 Random Access

When you access a file randomly by key, each I/O request must contain the KEY or KEYFROM option. The contents of the specified key depend 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 record number relative 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 PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, you have the option of requesting that the READ statement match any record with an equal or greater key value, or any record with a greater key value.

6.2.3 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.

6.2.4 Block Input/Output

A block is a physical extent on a disk volume. You can elect to perform all of your I/O operations in terms of disk blocks: the file system transfers blocks of data at a time to your program. Your program is responsible for interpreting and deblocking the data in the blocks.

To perform block I/O, specify the BLOCK_IO option of the ENVIRONMENT attribute. This option is valid for disk files and for magnetic tape files. You can access a disk file for block I/O using either sequential or random access. If you are performing sequential block I/O, blocks are transferred to your program in sequential order. If you use keyed access, you must specify a virtual block number in the KEY or KEYFROM option. The first block in the file is number 1, the second block is number 2, and so on.

Block I/O to magnetic tape files can only be performed sequentially.

When a file is opened for block I/O, the SPACEBLOCK built-in subroutine can be used to move the file forward or backward a given number of blocks.

6.2.5 Access by Record Identification

You can access records in any type of disk file that is opened for any type of access using a record identification. You must do the following:

1. Specify the RECORD_ID_ACCESS option in the ENVIRONMENT attribute for the file.
2. Supply a 2-element array variable to save the record identification of records you want to access by record identification.
3. Specify the array variable in the RECORD_ID_TO option on a READ, REWRITE, or WRITE statement that accesses a record in the file, and save the value that is returned.
4. Specify the array variable that contains a valid record identification in a RECORD_ID option on a READ, REWRITE, or DELETE statement.

For example:

```
DECLARE LIBFILE FILE ENVIRONMENT(RECORD_ID_ACCESS),
        SAVED_ID(2) FIXED BINARY(31);

OPEN FILE (LIBFILE) DIRECT;
READ FILE (LIBFILE) KEY(MODULE_NAME) INTO (INREC)
           OPTIONS (RECORD_ID_TO(SAVED_ID));
```

The READ statement in this example returns the value of the record identification associated with the record whose key is indicated by the variable MODULE_NAME. The value of SAVED_ID may subsequently be used in a REWRITE or DELETE statement to modify the record.

6.3 Record Formats

VAX Record Management Services (RMS) allows the following types of record formats:

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

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.

6.3.1 Fixed-Length Records

In a file containing fixed-length records, all records have the same length. Thus, 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 the ENVIRONMENT attribute (see Chapter 7) to process records in the standard PL/I manner.

6.3.2 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.

Variable length is the default record format for PL/I for OpenVMS VAX and PL/I for OpenVMS AXP files. You can specify the MAXIMUM_RECORD_SIZE option to specify the maximum length that any record can have. For example:

```
DECLARE VAR_FILE FILE RECORD OUTPUT
        ENVIRONMENT (
                     MAXIMUM_RECORD_SIZE(80));
```

This declaration indicates that the file VAR_FILE has variable-length records, each with a maximum length of 80 characters.

When a file that has variable-length records is processed by READ statements, the file system checks the length of the variable specified in the INTO option to see if it is large enough to hold the record being read. If not, the ERROR condition is signaled.

6.3.3 Variable-Length Records with a Fixed-Length Control Area

Variable-length records with a fixed-length control area are similar to variable-length records. However, they also contain a fixed-length control field. The control field size is the same for every record in the file.

The fixed control area in a record can contain data that is not related to the other data in the file. For example, a line-oriented editing program could use the fixed control area for sequence numbers.

To create a file of variable-length records with a fixed-length control area, specify the FIXED_CONTROL_SIZE option of the ENVIRONMENT attribute. The length of the control area you specify is a permanent attribute of the file; it cannot be changed later.

When you specify the maximum record length for a file containing variable-length with fixed-length control records, the length you specify does not include the length of the fixed control area. For example:

```
DECLARE VFC_FILE FILE RECORD
        ENVIRONMENT (
                    MAXIMUM_RECORD_SIZE(250),
                    FIXED_CONTROL_SIZE(2));
```

When the file VFC_FILE is opened for writing, the size of the fixed-length control area is set at 2 bytes; the maximum size of the data portion of any record is set at 250 bytes.

I/O operations on the file can process the fixed control area. The READ statement returns the contents of the fixed control area to a variable specified in the FIXED_CONTROL_TO option. The REWRITE and WRITE statements modify the fixed control area using the variable specified in the FIXED_CONTROL_FROM option, if any.

If this file is printed by the OpenVMS system, the contents of the fixed control area are printed to the left of each record.

6.4 Carriage Control

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP provide a default carriage control for files that will be printed. This format, called carriage return format, can be specified in the ENVIRONMENT option list (see Chapter 7) 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 alternative form of carriage control is the PRINTER_FORMAT option of the ENVIRONMENT attribute, which 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.

6.5 Sequential Files

This section shows examples of some typical sequential file I/O operations on sequential disk files and on sequential devices, including magnetic tapes.

6.5.1 Creating a Sequential File

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

Appending Records to an Existing File

You can open a file with the APPEND option of the ENVIRONMENT attribute to add new records to 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.

Superseding an Existing File

The ENVIRONMENT option SUPERSEDE lets you create a new version of a file each time you write it, simultaneously deleting an existing version. By default, each time a specific file is written, PL/I for OpenVMS VAX or PL/I for OpenVMS AXP 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.

6.5.2 Using Magnetic Tape Files

Before you execute a PL/I program that reads or writes a sequential file on a magnetic tape volume, you must use the following sequence of DCL commands:

1. Use the ALLOCATE command to allocate a device on which to mount the tape volume. For example:

   ```
   $ ALLOCATE MT:
     _MTA0:  ALLOCATED
   ```

   The ALLOCATE command responds with the name of the physical device. You can now place the physical tape reel on the device.

2. If the tape is a new tape that you are going to write or overwrite, use the INITIALIZE command to format the tape and write a label on it. For example:

   ```
   $ INITIALIZE MTA0:  MYTAPE
   ```

   This command writes the label MYTAPE on the tape volume mounted on MTA0: (the system printer device).

3. Use the MOUNT command to ready the volume for use and, at your option, to define a logical name for the device and file. For example:

   ```
   $ MOUNT MTA0: MYTAPE TAPEFILE
   ```

After this sequence of commands, a PL/I program that writes records using the logical name TAPEFILE will be writing to the MTA0: tape volume. For example:

```
DECLARE OUTFILE FILE RECORD OUTPUT;
OPEN FILE(OUTFILE) TITLE('TAPEFILE:TAPE1.FIL');
```

When this OPEN statement is executed, the logical name TAPEFILE is translated to its equivalence MTA0:, and the file MTA0:TAPE1.FIL;0 is created. Magnetic tape files always have a version number of 0.

Format of Magnetic Tapes

RMS supports the magnetic tape structure defined by ANSI X3.27-1977, the Magnetic Tape Labels and File Structure for Information Interchange. The tapes are encoded in ASCII format and can be processed on 9-track tape drives only.

The INITIALIZE command writes a label on the tape in the required format. A tape created on an OpenVMS system can be read on another system that supports the same tape label format.

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 after 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.

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

The BLOCK_SIZE option, if specified, is ignored when the output file is not a magnetic tape device. Thus, if you create and test a program that writes to a magnetic tape, you can test the program's output on a disk device; PL/I ignores the block size.

When a blocked magnetic tape file is read, RMS determines the block size from the tape itself, and deblocks the file as individual records are read.

Performing Block Input/Output

If a tape file is opened with the ENVIRONMENT option BLOCK_IO, the file can be read or written in terms of blocks. The size of a block is determined by the value of the BLOCK_SIZE option. Each I/O operation transfers the specified number of bytes. For example:

```
OPEN FILE (TAPEFILE) ENV (BLOCK_IO,BLOCK_SIZE(2048));
READ FILE (TAPEFILE) INTO (READBUF);
```

These statements open the file TAPEFILE and read a 2048-byte block of data into the program variable READBUF. When a file is opened with the BLOCK_IO option, RMS does not block or deblock records. Your program must perform all necessary blocking and deblocking.

When a file is opened for block I/O, the program can advance the tape or move it backward a specified number of blocks by calling the SPACEBLOCK built-in subroutine. For example:

```
CALL SPACEBLOCK(TAPEFILE,3);
```

This call advances the file TAPEFILE forward three blocks. The SPACEBLOCK built-in subroutine is described in Chapter 9.

Multivolume Tape Files

The ANSI standard X3.27-1977 for magnetic tapes allows any of the following combinations of tape files:

- A single file on a single volume (that is, a reel)
- A single file on more than one volume
- Multiple files on a single volume
- Multiple files on more than one volume

When more than one tape volume is required to contain a file or files, the tapes constitute a volume set. The OpenVMS system processes volume sets as follows:

- When a file is being created on a tape volume and the tape reaches its end-of-volume, the magnetic tape control program (ACP) sends a message to a designated system operator requesting the operator to mount another tape volume. The program that is attempting to write to the tape must wait until the operator (or user who is performing operator functions) responds to the request. The response generally includes the initialization of another tape,

with a volume number one greater than the volume number of the current 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.

Normally, RMS requests new volumes automatically. However, 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 NEXT_VOLUME built-in subroutine. The NEXT_VOLUME subroutine is described in Chapter 9.

The physical process of volume switching, whether the switching is performed automatically by RMS or as a result of a call to the NEXT_VOLUME 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 *OpenVMS DCL Dictionary*.

6.5.3 Allocated and Spooled Devices

The OpenVMS system spools low-speed I/O devices such as printers by accumulating data for the device in a file, and then queuing 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 SS$_DEVALLOC status code.
- 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$ALLOC to allocate a device. For information on commands for device allocation and control, see the *OpenVMS DCL Dictionary*. For information on allocating devices using the SYS$ALLOC system service, see the *OpenVMS System Services Reference Manual*.

6.6 Relative Files

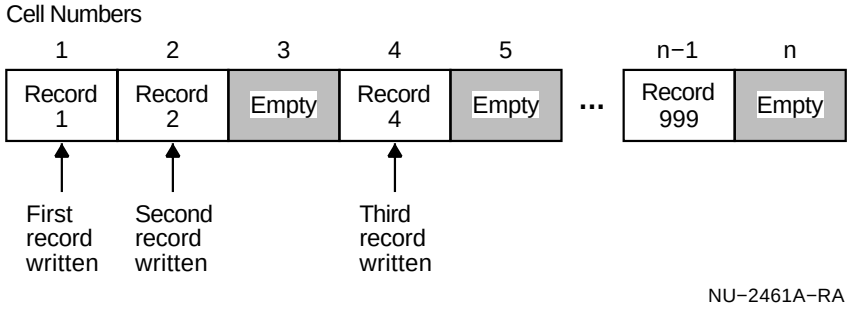
This section describes the organization of a relative file, suggests considerations for creating and using relative files, and shows examples of some typical relative file I/O operations.

6.6.1 Relative File Organization

The relative file organization is suitable for files with data that can be arranged serially and be 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, called the relative record number, is the key by which records are written and accessed.

Figure 6–1 illustrates a relative file in which not all cells contain records. The first record written to the file was relative record number 1 (which may have been data for a part numbered 1 or an employee whose number is 1, for example). The second record written was relative record number 2. The third record written was relative record number 4; thus cell number 3 does not contain a record.

Figure 6–1 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.

6.6.2 Creating a Relative File

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, relative file organization is the default organization for files that are opened with the KEYED attribute. Thus, when a WRITE statement is directed to a file with the KEYED and OUTPUT attributes, PL/I for OpenVMS VAX or PL/I for OpenVMS AXP creates a relative file.

When you initially create a relative file in a PL/I program, you should give consideration to using the following ENVIRONMENT options to maximize the efficiency of I/O operations on the file:

- MAXIMUM_RECORD_NUMBER
- MAXIMUM_RECORD_SIZE
- BUCKET_SIZE
- FILE_SIZE
- EXTENSION_SIZE

Considerations for specifying values for each of these follow. For more detailed information on file design, see the *Open VMS Record Management Services Manual*.

Maximum Record Number

The MAXIMUM_RECORD_NUMBER option specifies the largest relative record number that will be used in the file, and thus specifies the maximum number of cells that the file can have. If you do not specify a maximum record number, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP sets the maximum record number to zero. This permits the file to expand to any size.

For example, if a relative file will contain inventory data on 600 parts, and the part number is to be used as the file key, the MAXIMUM_RECORD_NUMBER option can be specified as follows:

```
DECLARE PARTS FILE ENVIRONMENT (
              MAXIMUM_RECORD_NUMBER (600));
```

You should realistically allow for future expansion of the file when you specify a maximum record number; the number is a permanent attribute of the file and cannot be changed. PL/I signals the KEY condition when a key value is too large.

Maximum 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:

- For fixed-length records:

  ```
  cell-size = 1 + record-size
  ```

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

- For 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.

- For variable-length records with fixed control:

  ```
  cell-size = 3 + maximum-record-size + size-of-fixed-control-area
  ```

  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, PL/I for OpenVMS VAX or PL/I for OpenVMS AXP uses the default length of 480 bytes.

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 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 by calculating whether space is being wasted. For example:

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

The file EMP_FILE will be created with 81-byte cells and buckets that have 2048 bytes (that is, four 512-byte blocks). Each bucket can contain twenty-five 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 7. That description contains formulas for calculating the bucket size within the limits required by RMS.

File Size

To avoid the processing overhead that occurs when a file is extended beyond an initial default allocation, you can specify the FILE_SIZE option when you create a relative file. If you specify this option, you can preallocate all the space that will ever be required for the cells in the file.

To determine the space requirements of a file, you can use the following formulas. To compute the size of the file in 512-byte blocks:

$$file\ size\ = \frac{511\ +\ (number\ of\ buckets\ *\ bytes\ per\ bucket)}{512}$$

To compute the number of buckets:

$$number\ of\ buckets\ = \frac{number\ of\ records}{number\ of\ cells\ per\ bucket}$$

You can also specify the CONTIGUOUS or CONTIGUOUS_BEST_TRY options in conjunction with the FILE_SIZE option to attempt to allocate all of the space for the file in contiguous disk blocks.

Extension Size

If you plan to add data to the file, you should specify a reasonable default extension size to reduce the number of times the file is extended. The FDL attribute FILE EXTENSION sets the extension size. The value to be assigned depends on the size of the records in the file and on the number of records that will be added to the file. The EDIT/FDL Utility calculates and assigns the correct extension size. For more information on the EDIT/FDL Utility, see Section 6.7.2.

You can also specify the extension size by using the DCL command SET FILE/EXTENSION=n. To determine the value to be assigned to the EXTENSION attribute, multiply the record size by the number of records to be added.

If you do not specify an extension size, RMS allocates the number of blocks needed for each extension.

6.6.3 Using Relative Files

You can create a relative file from any existing file that is suitable for relative file organization. Example 6–1 illustrates copying a sequential file with fixed-length records into a relative file.

Example 6–1 Creating a Relative File

```
COPY_TO_RELATIVE: PROCEDURE OPTIONS(MAIN);

    %INCLUDE PARTLIST;  /* Declaration of PARTLIST */  /* 1  */

    DECLARE OLDFILE FILE INPUT RECORD SEQUENTIAL;
    DECLARE RECORD_NUMBER FIXED BINARY;

    DECLARE
        PARTS FILE OUTPUT KEYED RECORD ENVIRONMENT(
                MAXIMUM_RECORD_NUMBER(600),          /* 2  */
                FIXED_LENGTH_RECORDS,
                MAXIMUM_RECORD_SIZE(36));

    ON ENDFILE(OLDFILE) BEGIN;                  /* 3  */
        CLOSE FILE(OLDFILE), FILE(PARTS);
        STOP;
        END;

    OPEN FILE(OLDFILE), FILE(PARTS);

    DO WHILE('1'B);
        READ FILE(OLDFILE) INTO(PARTLIST);  /* 4  */
        RECORD_NUMBER = PARTLIST.NUMBER;
        WRITE FILE(PARTS) FROM(PARTLIST) KEYFROM(RECORD_NUMBER); /* 5  */
        END;

    END COPY_TO_RELATIVE;
```

The following notes are keyed to Example 6–1:

1 The structure PARTLIST describes the layout of the records in the file. The records will be ordered in the relative file according to the field PARTLIST.NUMBER.

2 The relative file PARTS is declared with a maximum record number of 600. It has fixed-length, 36-byte records.

3 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 file is closed and the STOP statement terminates the program.

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.

Populating a Relative File

In Example 6–1, the file PARTS is opened with the KEYED and OUTPUT attributes. When this program is executed, 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 allowed for any record in the file is 600, but the largest record number specified for a record is 200, then RMS allocates only enough space 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 use either of the following techniques to allocate space for the entire file:

- Specify the FILE_SIZE option to allocate space for the file when it is created.
- 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, because RMS will not need to perform repeated extensions to the file as records are added.

Updating a Relative File

To add or modify records in a relative file, open the file with the DIRECT and UPDATE attributes. For example, a procedure that updates the file PARTS when new stock is ordered might contain the following:

```
ORDER_PARTS: PROCEDURE (ORDERED_AMOUNT,PART_NUMBER);
%INCLUDE PARTLIST;      /* Declaration of PARTLIST */
DECLARE (ORDERED_AMOUNT,PART_NUMBER) FIXED BIN(15);
DECLARE PARTS FILE RECORD DIRECT UPDATE;

       OPEN FILE(PARTS);
       READ FILE(PARTS) INTO(PARTLIST)
                        KEY(PART_NUMBER);
       PARTLIST.ON_ORDER = PARTLIST.ON_ORDER +
                           ORDERED_AMOUNT;
       REWRITE FILE(PARTS) FROM(PARTLIST);
       CLOSE FILE(PARTS);
END;
```

In this example, the procedure ORDER_PARTS receives as its parameters the order quantity and the part number. It reads the record associated with the part number from the file, adds the order quantity to the existing quantity, and rewrites the record.

Reading a Relative File Sequentially

You can access a relative file sequentially as well as by key. When you access the file sequentially, each READ statement returns the record in the next cell that contains a record, skipping empty cells. The following example illustrates reading a relative file sequentially:

```
PRINT_PART: PROCEDURE OPTIONS(MAIN);
%INCLUDE PARTLIST;      /* Declaration of PARTLIST */

DECLARE PARTS FILE,
        CHECK_NUM FIXED;

DECLARE EOF BIT(1) ALIGNED INIT('0'B);

        OPEN FILE(PARTS) INPUT SEQUENTIAL RECORD KEYED;
        ON ENDFILE(PARTS) EOF = '1'B;
```

```
        READ FILE(PARTS) INTO(PARTLIST) KEYTO(CHECK_NUM);
        DO WHILE (^EOF);
            PUT SKIP EDIT(PARTLIST.NAME,    /* Output data */
                      UNIT_PRICE,
                      IN_STOCK,
                      ON_ORDER)
                (A,X,A,X,A,X,A);
            PUT SKIP EDIT('Relative record number',CHECK_NUM,
               'Part number:',PARTLIST.NUMBER)
               (X(10),A,X,F(5),A,X,A);     /* Output
                                          verification */
            READ FILE(PARTS) INTO(PARTLIST) KEYTO(CHECK_NUM);
END;
```

This procedure outputs the contents of the file PARTS, listing each field in the data records described by PARTLIST. The READ statement specifies the KEYTO option; the procedure outputs the value returned to the variable CHECK_NUM.

6.6.4 Error Handling

PL/I signals the KEY condition when errors occur during the processing of 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 following sample ON-unit 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;
    %INCLUDE $RMSDEF;
    /* Check for duplicate records */
    IF ONCODE() = RMS$_REX        /* If duplicate */
    THEN DO;
       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 sets symbolic names for two specific status values returned by ONCODE, from PLI$STARLET:

- 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 can also return the values associated with the following status codes:

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

The symbolic names for these status codes are included from module $RMSDEF, from PLI$STARLET.TLB.

6.7 Indexed Sequential Files

This section describes the organization of indexed sequential files, suggests considerations for creating and using them, and shows examples of some typical operations.

6.7.1 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 (indexes) associated with the records. Each file must have a primary key—that 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.

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

Figure 6–2 illustrates an indexed sequential file with a single key.

The records in the file illustrated in Figure 6–2 consist of address data that might have been defined in a PL/I structure as follows:

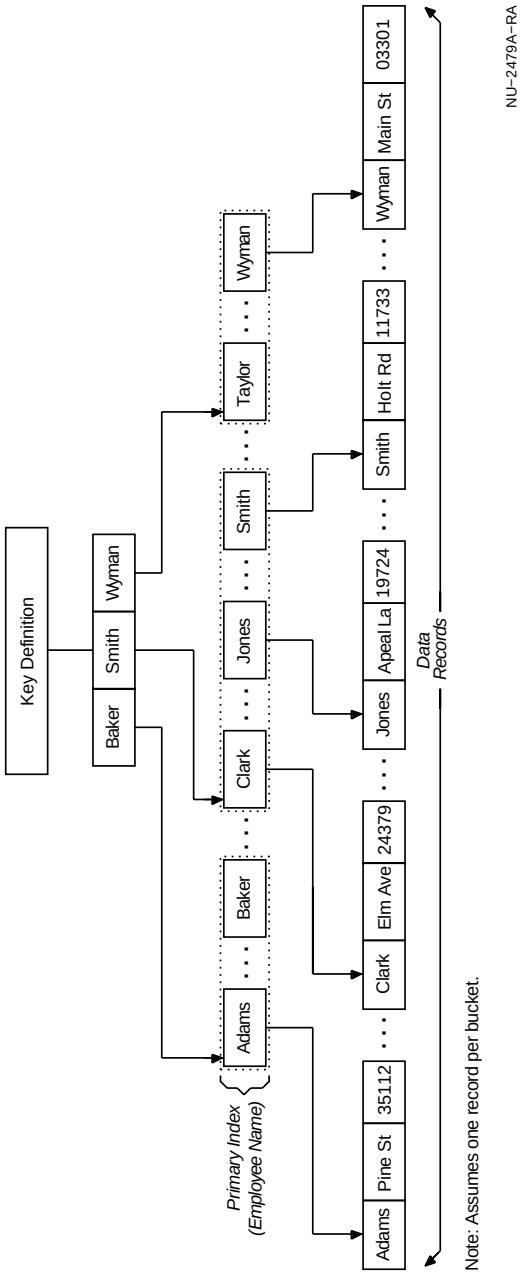
```
DECLARE 1
ADDRESS_FILE,
          2 EMPLOYEE_NAME CHARACTER(30),
          2 ADDRESS,
            3 STREET CHARACTER(20),
            3 ZIP_CODE CHARACTER(5);
```

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 values and location pointers. When records are accessed by key, RMS uses the tree to locate individual records. Thus, when a PL/I program accesses 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 knows only that it has added a record to the file.

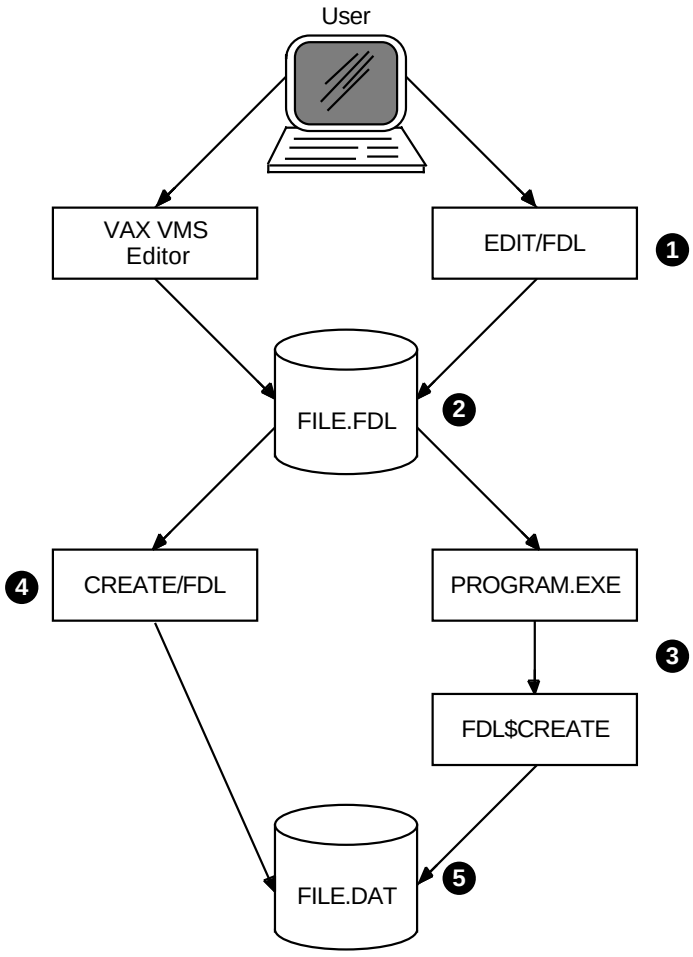
Figure 6–2 An Indexed Sequential File



6.7.2 Defining and Creating an Indexed Sequential File

As shown in Figure 6–3, an indexed sequential file must first be defined, then created. You define the characteristics of your indexed sequential file using the VMS File Definition Language (FDL), and then use that definition (the resulting FDL file) to create the indexed sequential file for use by your program.

Figure 6–3 Creating a Data File



An FDL definition file can be set up by several alternative methods. You can use any of the OpenVMS editors to create the file, or create it with a PL/I program, or use the FDL Facility itself. The following general steps are keyed to the callout numbers in Figure 6–3.

1 Use an editor to describe the data file.

2 An FDL language source file defines the data file.

3 You can use a program to create an FDL language source file and call FDL$CREATE to create a data file.

4 You can use FDL to create a data file.

5 The data file is created.

The following sections explain in detail how to create a definition file by using the FDL Facility and how to create one through a PL/I program.

6.7.2.1 Using EDIT/FDL

You can use the command EDIT/FDL to define an indexed sequential file for PL/I for OpenVMS VAX and PL/I for OpenVMS AXP. Note that while you can construct FDL file specifications using any of the OpenVMS text editors, the FDL Facility gives you the added convenience of systematic prompting for and automatic formatting of the specifications. The specifications are written in the File Definition Language (FDL) and the files are referred to as FDL files.

The FDL Facility is interactive: it prompts you to enter data and responds with error messages when you enter data incorrectly. It supplies many default values. The only data that you must specify is as follows:

- The file name of the file you are creating
- The word INDEXED, to indicate that the file is an indexed sequential file
- The number of records that will initially be loaded into the file
- The mean record size
- The size of the key

You can obtain help by pressing RETURN in response to the first menu. Each item on the menu leads to another menu in the specification of the data file. You can return to the original menu by pressing Ctrl/z.

```
$ EDIT/FDL Return
_File: STATE50 Return

                    Parsing Definition File

        DBA0:[SMITH]STATE50.FDL; will be created.

        Press return to continue (^Z for Main Menu) Return


                     VAX-11 FDL Editor

        Add     to insert one or more lines into the FDL definition
        Delete  to remove one or more lines from the FDL definition
        Exit    to leave the FDL Editor after creating the FDL file
        Help    to obtain information about the FDL Editor
        Invoke  to initiate a script of related questions
        Modify  to change existing line(s) in the FDL definition
        Quit    to abort the FDL Editor with no FDL file creation
        Set     to specify FDL Editor characteristics
        View    to display the current FDL definition

        Main Editor Function          (Keyword)[Help]:INVOKE Return
```

FDL displays further menus to guide you through the creation of the file.

After you define your data file, FDL displays a message like the following, giving the file specification of the definition file:

```
DBA0:[SMITH]STATE50.FDL;1    40 lines
```

FDL formats the information that you supply so that it can be used to create an indexed sequential file. However, you can also use EDT or any other OpenVMS text editor to design an FDL text.

FDL allows you to specify many characteristics for your file, but requires you to specify only a limited number. The following example, written in FDL, illustrates some of the possible file characteristics. The explanatory comments to the right would not appear in an FDL file; they have been added for your information.

```
SYSTEM
        SOURCE                  VAX/VMS

FILE
        NAME                    state50.dat /* file name */
        ORGANIZATION            indexed

RECORD
        CARRIAGE_CONTROL        carriage_return
        FORMAT                  fixed       /* record attributes */
        SIZE                    60
AREA 0
        ALLOCATION              25  /* number of blocks */
        BEST_TRY_CONTIGUOUS     yes /* as close as possible */
        BUCKET_SIZE             2   /* number of blocks in record */
        EXTENSION               2   /* blocks for extension */
AREA 1
        ALLOCATION              3
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             2
        EXTENSION               0
AREA 2
           .
           .
           .
AREA 7
        ALLOCATION              2
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             1
        EXTENSION               0
KEY 0                                   /* primary key */
        CHANGES                 no      /* cannot be changed */
        DATA_AREA               0       /* data in AREA 0 */
        DATA_FILL               100     /* %fill for data */
        DUPLICATES              no      /* not allowed */
        INDEX_AREA              1       /* index in AREA 1 */
        INDEX_FILL              100     /* %fill for index */
        LEVEL1_INDEX_AREA       1       /* level-1 index in AREA 1 */
        PROLOGUE                2       /* type for index */
        SEG0_LENGTH             20      /* key length */
        SEG0_POSITION           0       /* key position */
        TYPE                    string /* character string */
             .
             .
             .
KEY 3                                   /* third alternate key */
        CHANGES                 yes
        DATA_AREA               6
        DATA_FILL               100
        DUPLICATES              yes
        INDEX_AREA              7
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       7
        SEG0_LENGTH             3
        SEG1_LENGTH             3
        SEG2_LENGTH             3
        SEG0_POSITION           0
        SEG1_POSITION           20
        SEG2_POSITION           40
        TYPE                    string
```

After creating an FDL file with EDIT/FDL or a text editor, you can then use the command CREATE/FDL, which uses the specifications in the FDL file to create a new, empty data file. Thus, you can use EDIT/FDL to define the data file, and then create the data file when it is required later. See the *OpenVMS File Definition Language Facility Reference Manual* for further details.

6.7.2.2 Using a PL/I Program

You can write a PL/I program to define an indexed sequential file or to create an indexed sequential file from an existing definition, or both. The following lines in the program enable the optional generation of a file definition and the use of that definition to create a file:

```
%INCLUDE FDL$CREATE;
   .
   .
   .
CALL FDL$CREATE ('STATE50.FDL');
```

The following program, CREATE50, defines a file similar to the file defined with the FDL Facility in Section 6.7.2.1.

```
CREATE50: PROCEDURE OPTIONS(MAIN);
DECLARE ANSWER CHAR(1);  /* answer variable */
DECLARE (I,J) FIXED BIN(31);  /*index control variables */

%INCLUDE FDL$CREATE;

/* default allocation, bucket size, and extent for each area */
DECLARE AREA_ATTRIBUTES(0:7,0:2) FIXED BIN(7) STATIC INITIAL (
        25, 2, 2,
         2, 2, 0,
        15, 2, 1,
         3, 2, 0,
        20, 2, 2,
         3, 2, 0,
         9, 1, 0,
         7, 1, 0);

/* default areas, fills, segment lengths, and positions for each key */
DECLARE KEY_ATTRIBUTES(0:3,0:10) FIXED BIN(7) STATIC INITIAL (
        0, 100, 1, 100, 1, 20,  0,  0,  0,  0,  0,
        2, 100, 3, 100, 3, 20,  0,  0, 20,  0,  0,
        4, 100, 5, 100, 5, 20,  0,  0, 40,  0,  0,
        6, 100, 7, 100, 7,  3,  3,  3,  0, 20, 40);

/* FDL file that will be created if necessary */
DECLARE FDL FILE PRINT OUTPUT;

/* find out if FDL file already exists */

GET LIST (ANSWER)
     OPTIONS(PROMPT('Have you already created STATE50.FDL
         with EDIT/FDL or EDT? '));

/* file definition (.FDL) file creation */
IF ANSWER = 'N' | ANSWER = 'n'
THEN
    DO;

    PUT SKIP LIST ('  Creating STATE50.FDL.....');

    OPEN FILE (FDL) TITLE ('state50.fdl') PAGESIZE(32767);

    /* define system */

    PUT SKIP FILE (FDL) LIST ('SYSTEM');
    PUT SKIP FILE (FDL) LIST ('     SOURCE            VAX/VMS');

    /* define file */
    PUT SKIP FILE (FDL) LIST ('FILE');
    PUT SKIP FILE (FDL) LIST ('     NAME              state50.dat');
    PUT SKIP FILE (FDL) LIST ('     ORGANIZATION      indexed');

    /* define record */

    PUT SKIP FILE (FDL) LIST ('RECORD');
    PUT SKIP FILE (FDL) LIST ('     CARRIAGE_CONTROL  carriage_return');
    PUT SKIP FILE (FDL) LIST ('     FORMAT            fixed');
    PUT SKIP FILE (FDL) LIST ('     SIZE              60');

    /* define areas */
```

```
    DO I = 0 TO HBOUND (AREA_ATTRIBUTES,1);

        PUT SKIP FILE (FDL) LIST ('AREA '||trim(character(i)));
        PUT SKIP FILE (FDL) LIST ('     ALLOCATION      '||
                                  character(area_attributes(i,0)));
        PUT SKIP FILE (FDL) LIST ('     BEST_TRY_CONTIGUOUS  yes');
        PUT SKIP FILE (FDL) LIST ('     BUCKET_SIZE     '||
                                  character(area_attributes(i,1)));
        PUT SKIP FILE (FDL) LIST ('     EXTENSION       '||
                                  character(area_attributes(i,2)));

        END;

    /* define keys */
    DO I = 0 TO HBOUND (KEY_ATTRIBUTES,1);

        PUT SKIP FILE (FDL) LIST ('KEY '||TRIM(CHARACTER(I)));

        IF I = 0 THEN PUT SKIP FILE (FDL) LIST ('     CHANGES                 no');
                 ELSE PUT SKIP FILE (FDL) LIST ('     CHANGES                 yes');

        PUT SKIP FILE (FDL) LIST ('     DATA_AREA           '||
                                  CHARACTER(KEY_ATTRIBUTES(I,0)));
        PUT SKIP FILE (FDL) LIST ('     DATA_FILL           '||
                                  CHARACTER(KEY_ATTRIBUTES(I,1)));
        IF I = 0 THEN PUT SKIP FILE (FDL) LIST ('     DUPLICATES              no');
                 ELSE PUT SKIP FILE (FDL) LIST ('     DUPLICATES              yes');

        PUT SKIP FILE (FDL) LIST ('     INDEX_AREA          '||
                                  CHARACTER(KEY_ATTRIBUTES(I,2)));
        PUT SKIP FILE (FDL) LIST ('     INDEX_FILL          '||
                                  CHARACTER(KEY_ATTRIBUTES(I,3)));
        PUT SKIP FILE (FDL) LIST ('     LEVEL1_INDEX_AREA   '||
                                  CHARACTER(KEY_ATTRIBUTES(I,4)));

        DO J = 0 TO 2;
            IF KEY_ATTRIBUTES(I,5+J*2) ^= 0
            THEN DO;
                 PUT SKIP FILE (FDL) LIST ('     SEG'||TRIM(CHARACTER(J))||'_LENGTH       '||
                                           CHARACTER(KEY_ATTRIBUTES(I,5+J*2)));
                 PUT SKIP FILE (FDL) LIST ('     SEG'||trim(character(j))||'_POSITION     '||
                                           CHARACTER(KEY_ATTRIBUTES(I,5+J*2)));
               END;
             END;

        PUT SKIP FILE (FDL) LIST ('     TYPE                    string');

        END;

        CLOSE FILE (FDL);
     END;
/* create STATE50.DAT using STATE50.FDL as the definition file */

PUT SKIP LIST ('  Creating file using STATE50.FDL.....');

CALL  FDL$CREATE ('STATE50.FDL');

END CREATE50;
```

This program uses the same FDL statements that would be used if you had issued the EDIT/FDL command. Notice that the program asks you whether you have already created the file and, if so, calls FDL$CREATE, which permits you to use a preexisting file definition.

Note that the following command:

```
$ CREATE/FDL=STATE50
```

is equivalent to the following command and response:

```
$ RUN CREATE50
Have you already created STATE50.FDL with EDIT/FDL or EDT? Y
```

6.7.3 Defining Keys

An indexed sequential file must have at least one key. The first (primary) key is always numbered 0. An indexed sequential file can have up to 255 keys; however, for file-processing efficiency it is recommended that you define no more than 7 or 8 keys. (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, however, is unaffected by the number of keys.)

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

- Position and size
- Data type
- Index number
- Options selected

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.

Specifying Key Position and Size

When you specify a key, you must specify both its position in the record and its length. The position is 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.

If the ENVIRONMENT option SCALARVARYING is in effect, the key size for a CHARACTER VARYING key should be 2 more than your declared maximum size; you specify a key position offset 2 from the variable base offset.

To determine the key positions for fields within a structure, you can examine the storage map in the program listing that defines the structure. The following structure definition illustrates the relationships between key field definitions and the storage map offsets:

```
1      FOO: PROCEDURE;
2    1 DECLARE 1 STATE BASED (STATE_PTR),
3    1             2 NAME CHARACTER (20), /*Primary key */ 1
4    1             2 POPULATION FIXED BINARY(31), /*3rd alternate key */4
5    1             2 CAPITAL,
6    1                 3 NAME CHARACTER(20),
7    1                 3 POPULATION FIXED BINARY(31),
8    1             2 LARGEST_CITIES(2),
9    1                 3 NAME CHARACTER(30),
10   1                 3 POPULATION FIXED BINARY(31),
11   1             2 SYMBOLS,
12   1                 3 FLOWER CHARACTER(30), /*1st alternate key*/ 2
13   1                 3 BIRD CHARACTER(30); /*2nd alternate key*/ 3
14   1
15   1 END FOO;
```

The resultant storage map shows the following:

1 The primary key is 20 bytes, offset 0 bytes from the base of the structure or record.

2 The first alternate key is 30 bytes, offset 116 bytes from the base of the structure or record.

3 The second alternate key is 30 bytes, offset 146 bytes from the base of the structure or record.

4 The third alternate key is four bytes, offset 20 bytes from the base of the structure or record.

The storage map is:

```
          +-------------+
          | Storage Map |
          +-------------+

External Entry Points and Variables Declared Outside Procedures
---------------------------------------------------------------

 Identifier Name                Storage     Size     Line  Attributes
 ---------- ----                -------     ----     ----  ----------

 FOO                                                 1     ENTRY, EXTERNAL
```

```
Procedure FOO on line 1
-----------------------

 Identifier Name                Storage     Size     Line  Attributes
 ---------- ----                -------     ----     ----  ----------

 BIRD                                       30 BY    13    OFFSET FROM BASE IS 146 BY,
                                                           MEMBER OF STRUCTURE SYMBOLS,
                                                           CHARACTER(30) UNALIGNED, NONVARYING 3

 CAPITAL                                    24 BY    5     OFFSET FROM BASE IS 24 BY,
                                                           MEMBER OF STRUCTURE STATE, STRUCTURE

 FLOWER                                     30 BY    12    OFFSET FROM BASE IS 116 BY,
                                                           MEMBER OF STRUCTURE SYMBOLS,
                                                           CHARACTER(30) UNALIGNED, NONVARYING 2

 LARGEST_CITIES                             68 BY    8     OFFSET FROM BASE IS 48 BY,
                                                           MEMBER OF STRUCTURE STATE, STRUCTURE
                                                           DIMENSION

 NAME                                       30 BY    9     OFFSET FROM BASE IS 48 BY, MEMBER OF
                                                           STRUCTURE LARGEST_CITIES
                                                           CHARACTER(30), UNALIGNED, NONVARYING

 NAME                                       20 BY    6     OFFSET FROM BASE IS 24 BY, MEMBER OF
                                                           STRUCTURE CAPITAL, CHARACTER(20)
                                                           UNALIGNED, NONVARYING

 NAME                                       20 BY    3     OFFSET FROM BASE IS 0 BY, MEMBER OF
                                                           STRUCTURE STATE, CHARACTER(20)
                                                           UNALIGNED, NONVARYING 1

 POPULATION                                 4 BY     10    OFFSET FROM BASE IS 78 BY, MEMBER OF
                                                           STRUCTURE LARGEST_CITIES
                                                           FIXED BIN(31,0), ALIGNED, PRECISION

 POPULATION                                 4 BY     7     OFFSET FROM BASE IS 44 BY, MEMBER OF
                                                           STRUCTURE CAPITAL, FIXED BIN(31,0)
                                                           ALIGNED, PRECISION

 POPULATION                                 4 BY     4     OFFSET FROM BASE IS 20 BY, MEMBER OF
                                                           STRUCTURE STATE, FIXED BIN(31,0)
                                                           ALIGNED, PRECISION
4
 STATE                          based       176 BY   2     STRUCTURE

 SYMBOLS                                    60 BY    11    OFFSET FROM BASE IS 116 BY, MEMBER OF
                                                           STRUCTURE STATE, STRUCTURE

   .
   .
   .
```

After you enter the EDIT/FDL command, you can specify the keys to FDL as follows:

```
        Enter Desired Primary       (Keyword)[-]   : KEY 0

                - Legal KEY 0 Secondary Attributes -

CHANGES                 yes/no    LEVEL1_INDEX_AREA   number
DATA_AREA               number    NAME                string
DATA_FILL               number    NULL_KEY            yes/no
DATA_KEY_COMPRESSION    yes/no    NULL_VALUE          char/num
DATA_RECORD_COMPRESSION yes/no    POSITION            number
DUPLICATES              yes/no    PROLOGUE            number
INDEX_AREA              number    TYPE                keyword
INDEX_COMPRESSION       yes/no    SEGn_LENGTH         number
INDEX_FILL              number    SEGn_POSITION       number
LENGTH                  number

   Enter KEY 0 Attribute      (Keyword)[-]   :POSITION Return
```

```
KEY 0
       SEG0_POSITION

Enter value for this Secondary  (0-16299)[-]   :0 Return

                       - Resulting Primary Section -

KEY 0
       SEG0_POSITION          0

Continue with this Same Primary (Yes/No)[No]   :YES Return
```

At this point, the menu Secondary Attributes reappears on your screen:

```
Enter KEY 0 Attribute       (Keyword)[-]   :LENGTH Return

KEY 0
        SEG0_LENGTH
Enter value for this Secondary  (0-255)[-]  :20 Return

                       - Resulting Primary Section -

KEY 0
        SEG0_LENGTH           20
        SEG0_POSITION         0

Continue with this Same Primary (Yes/No)[No]  : Return
```

At this point, you return to the first menu. To establish the keys necessary for ADDRESS.DAT, you need to step through the entire process again. In response to the prompt after the first menu, type ADD; in response to the prompt after the second menu, type KEY 1. Give KEY 1 (for example) a position of 116 and a length of 30.

Establish the third and last key the same way. Give KEY 2 a position of 146 and a length of 30. Then go back to the first menu and type VIEW in response to the prompt. A summary appears as follows:

```
KEY 0
       SEG0_LENGTH           20
       SEG0_POSITION         0

KEY 1
       SEG0_LENGTH           30
       SEG0_POSITION         116

KEY 2
       SEG0_LENGTH           30
       SEG0_POSITION         146
```

Use FDL to change information if necessary.

Key Data Types

Table 6–2 summarizes the valid data types for keys in 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 FDL Facility.

Table 6–2 Key Data Types

Data Type	PL/I Declaration	FDL Specification
String[1]	CHAR(n), where 1 < n < 255	STR n
15-bit signed integer	FIXED BINARY(15)	INT 2
31-bit signed integer	FIXED BINARY(31)	INT 4
63-bit signed integer	FIXED BINARY(63)	INT 8
16-bit unsigned binary[2]	FIXED BINARY(15)	BIN 2
32-bit unsigned binary[2]	FIXED BINARY(31)	BIN 4
64-bit unsigned binary[2]	FIXED BINARY(64)	BIN 8
Packed decimal	FIXED DECIMAL(n) where 1< n < 16	DECIMAL n

[1] PL/I for OpenVMS VAX and PL/I for OpenVMS AXP support segmented character-string keys.

[2] PL/I does not distinguish between signed and unsigned integers. Thus, the difference between signed integer keys and unsigned binary keys is in the key collating sequence. For signed integer keys, the collating sequence is from the lowest negative number to the highest positive number (for example –32768, –32767, ... 0, 1, 2, ... 32767). For unsigned binary keys, the collating sequence is from zero to the highest positive number, then from the lowest negative number to –1 (for example 0, 1, 2, ... 32766, 32767, –32768, –32767, ... –1).

Index Numbers

When you define the keys in an indexed sequential file, you must assign an index number to each alternate key. The index number of the primary key is always 0; subsequent alternate key indexes are numbered 1, 2, and so on. When you create a file with the FDL Facility, you must define the keys in index number order.

When you want to access a record in an indexed sequential file by an alternate key, you specify the index number. You can specify the index number either in the INDEX_NUMBER option of ENVIRONMENT or in the INDEX_NUMBER option of a record I/O statement.

For example, to access the record for a state whose flower is MAGNOLIA in the indexed file STATE_FILE, when the current index is not 1, you must specify the index number 1, as in the following example:

```
READ FILE(STATE_FILE)
INTO(STATE) KEY('MAGNOLIA')
        OPTIONS(INDEX_NUMBER(1));
```

This READ statement reads the first record whose key in the first alternate index is MAGNOLIA and sets the current index number to 1.

Key Options

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

- 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 the alternate index.

These options are described in the *OpenVMS File Definition Language Facility Reference Manual*.

6.7.4 Using Indexed Sequential Files

After you have defined and created an indexed sequential file, 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.

Note

When PL/I copies the KEYFROM value into the record, it overwrites anything already in those positions, while distributing segmented values as specified by the RMS key description. Therefore, it is important that the key value come from some variable other than the record variable itself.

Populating an Indexed Sequential File

There are two techniques for optimizing the initial population (loading) of an indexed sequential file:

- Use the RMS CONVERT utility to load the file. This utility copies records from an existing file to load the indexed file. This utility also optimizes the index structure.

- Use the INITIAL_FILL option of the ENVIRONMENT attribute. This option causes RMS to apply the fill number specified for index buckets in the file when the file was created. This option is effective only if a fill number was specified to the FDL Facility. It leaves unused space in the file for the key values that are inserted. Subsequent insertion of records with similar key values (without the INITIAL_FILL option) will tend to use the free space provided within the file, without causing further growth of the file.

For details on specifying fill numbers for the FDL Facility, see the *OpenVMS File Definition Language Facility Reference Manual*.

Reading an Indexed Sequential File Sequentially

To read records in an indexed sequential file in collating order by key value, open the file with the INPUT and SEQUENTIAL attributes. The following example illustrates reading the file STATE_FILE in sequential order using the primary key, that is, using the STATE.NAME field. This procedure uses the SET option of the READ statement; thus, no space is required in the records.

```
DECLARE STATE_PTR POINTER,
        STATE_FILE FILE,
        EOF BIT(1) INITIAL ('0'B);
DECLARE 1 STATE BASED (STATE_PTR),
          2 NAME CHARACTER(20),
          .
          .
          .
ON ENDFILE(STATE_FILE) EOF = '1'B;
OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
READ FILE(STATE_FILE) SET(STATE_PTR);
DO WHILE (^EOF);
    PUT SKIP(2) LIST('State:',STATE.NAME);
    PUT SKIP(2) EDIT('Population:',STATE.POPULATION)
                        (A,P'ZZ,ZZZ,ZZZ');
      .
      .
      .
    READ FILE(STATE_FILE) SET(STATE_PTR);
    END;
```

In this example, the procedure reads the records in the file STATE_FILE beginning with the first record in its primary index, the NAME field.

Accessing Records by Alternate Key

To use an alternate key to read a record in an indexed sequential file, specify the INDEX_NUMBER option on the READ statement.

For example, if a file containing data about states has as its primary key the state name, it might have alternate keys for state flowers, birds, and so on. Assuming that a field called FLOWER is the first alternate key, you could access the record for a state whose flower is MAGNOLIA by writing the following statements:

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

The INDEX_NUMBER option followed by 1 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 the ENVIRONMENT attribute as in the following 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 on the basis of its second alternate index, the BIRD field.

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 the 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,NEWFLOWER) 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(NEWFLOWER) OPTIONS(
                 PROMPT('New state flower name: '));
    STATE.FLOWER = NEWFLOWER;
    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.

Specifying the Type of Key Match

RMS allows generic key matching; that is, it can locate a record in an indexed sequential file whose key is greater than a specified key value, or whose key is greater than or equal to a specified key value. In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, you can access records by generic key using the MATCH_NEXT and MATCH_NEXT_EQUAL options on a record I/O statement.

For example, the file STATE_FILE's third alternate key is the state's population. To list all states whose populations are greater than a specified size, you could open and read the file as follows:

```
DECLARE EOF BIT(1) STATIC INIT('0'B),
        SIZE FIXED BIN(31);
```

```
OPEN FILE(STATE_FILE) KEYED SEQUENTIAL ENV(
        INDEX_NUMBER(3));
GET LIST(SIZE) OPTIONS(PROMPT('Starting population: '));
READ FILE(STATE_FILE) SET (STATE_PTR) KEY(SIZE)
                OPTIONS(MATCH_NEXT_EQUAL);
DO WHILE (^EOF);
    PUT SKIP EDIT(STATE.NAME,'Population is ',
                        STATE.POPULATION)
                (A,A,P'ZZ,ZZZ,ZZZ');
    READ FILE(STATE_FILE) SET (STATE_PTR);
    END;
```

In this example, the size is obtained by a GET statement. The procedure opens the file, specifying the third alternate index, the POPULATION field, in the INDEX_NUMBER option. After accessing a record by matching the first key in this index that equals or exceeds the size entered, the procedure reads the file sequentially to the end-of-file, using that index.

Segmented Keys

PL/I supports the RMS segmented key concept. Segmented keys are used when a single key cannot be made in contiguous fields of the record. The key's parts are split up over the record, or reordered within a contiguous part of the record. Segmenting the key has the advantage of using only one key to represent several fields that are always defined and sorted as a conceptual unit.

When there is a KEYFROM clause with a segmented key, PL/I takes its usual action of copying the key value into the record variable and then writing the variable directly to disk through RMS.

PL/I copies the one contiguous value to the position or positions specified by the RMS definition for the segmented key. This means it must break up the value into subfields or segments. Those fields are determined by the RMS segment sizes for the key. Thus, for a 4-segment key of 8 bytes and individual sizes of 1, 1, 4, and 2, respectively, it would break up the one contiguous key value ABCDEFGH as follows:

```
Segment 0 = "A"
Segment 1 = "B"
Segment 2 = "CDEF"
Segment 3 = "GH"
```

Then it copies each segment into its correct position in the record—the positions as specified by RMS, not your record variable structure declaration.

When RMS orders the records by the key, it reconstructs the original key value as one piece and sorts it that way. Thus the relative ordering of the segments within the record has no bearing on how the original value is sorted.

These actions are the same for nonsegmented keys in PL/I. They are treated as keys of one segment only, of the appropriate RMS-determined data type.

If you need to use segmented keys, be aware of the following considerations and constraints:

- PL/I for OpenVMS VAX and PL/I for OpenVMS AXP support segmented keys for character-string data only. Do not use segmented keys for other data types.
- RMS requires the key values to be embedded in the record, rather than separate from it.

- There is a difference in the actions of the KEY option and the KEYFROM option that becomes important if you are using segmented keys, as follows:

  KEY(v)
  Indicates that an existing record is to be accessed at the indicated key value (v). There is no movement of the key value. (The KEY option is valid on the READ and REWRITE statements.)

  KEYFROM(v)
  Indicates that a new record is to be created at the position specified by the key value (v). Because RMS requires the key value to be embedded in the record, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP move the value into the correct place in the record buffer that is specified by the FROM option of the WRITE statement.

  Note that the key specified by the KEYFROM value is always the primary key, and that PL/I sets the key number to the primary index number (0). For segmented keys, PL/I copies the value into the correct key field positions.

  Note also that the KEYFROM option is used on the WRITE statement only, not the REWRITE statement. When you use a REWRITE statement, you need to ensure that the key value is in the record by first using a READ statement.

- RMS directly supports multikeyed files, but PL/I does not. Whereas in PL/I you would expect that only the INDEX_NUMBER key of reference would have its value filled in by the KEYFROM option, RMS actually writes all the keys in the record when it does a write or update operation. Thus, you must yourself fill in the unreferenced keys in the record before you use a WRITE or REWRITE statement.

Error Handling

PL/I signals the KEY condition when errors occur while keys for indexed sequential files are being processed. For example, if a key value specified on a READ statement specifies a key that does not exist, or if a WRITE statement attempts to write a record using a primary key value that already exists, the KEY condition is signaled.

You can write an ON-unit to detect and correct some of the more common key errors. For an example of an ON-unit that detects whether a record with a given key value was not found or whether an attempt was made to write a record whose key duplicates the value of an existing key, see Chapter 4.

7

Options of the ENVIRONMENT Attribute

The options of the ENVIRONMENT attribute provided by PL/I for OpenVMS VAX and PL/I for OpenVMS AXP let you do the following:

- 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 of the ENVIRONMENT attribute correspond directly to VAX Record Management Services (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 description of each option. The descriptions of the ENVIRONMENT options are arranged in alphabetical order. The chapter concludes with a discussion of the ENVIRONMENT options designed for file protection, file sharing, and I/O optimization.

7.1 Specifying and Using ENVIRONMENT Options

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

7.1.1 Arguments for ENVIRONMENT Options

ENVIRONMENT options fall into the following categories, based on whether they require an argument and, if so, what type of argument is required:

- Options that require you to specify an expression representing a value to override a default value provided by PL/I for OpenVMS VAX and PL/I for OpenVMS AXP
- Options that require you to provide a reference to a variable that either contains information pertinent to opening the related file or that will receive information when the related file is opened
- Options that can be specified with a Boolean expression that enables or disables the option (if no value is specified with an option, the option is enabled)

All arguments must be specified in parentheses following the name of the ENVIRONMENT option. For example:

```
ENVIRONMENT (
             MAXIMUM_RECORD_SIZE(1024),
             FILE_ID_TO(WORKFILE_ID),
             FIXED_LENGTH_RECORDS('1'B) )
```

Considerations for specifying each type of argument are given in the following sections.

7.1.1.1 Expressions

You can use integer expressions and character expressions in expression arguments for ENVIRONMENT options. The ways you can specify these arguments differ for DECLARE statements and for OPEN and CLOSE statements.

In a DECLARE statement, you must specify a constant expression. Integer expressions can consist of integer constants, constant identifiers defined by %REPLACE statements, and the operators +, –, *, and /. You must specify character expressions using character-string constants.

In an OPEN or a CLOSE statement, you can specify the argument using constant expressions or variable references, or expressions or variable references of the required type.

If a single variable is specified for an expression, its data type must be convertible to the data type of the option. All integer constants and expressions are converted to FIXED BINARY.

You can specify all character-string expressions using varying or nonvarying strings. The description of the option specifies the maximum length of a string argument.

For any of these options, PL/I applies a default value if no option is specified for the file when it is opened.

7.1.1.2 Variable References

Options that are specified by variable references cannot be specified in a DECLARE statement. The data type of the variable must match the data type described in the option description.

7.1.1.3 Boolean Values

For an option that can be enabled or disabled, you can specify a Boolean constant, that is, '1'B (to enable) or '0'B (to disable) the option in a DECLARE statement. In an OPEN or CLOSE statement, you can specify a Boolean constant, variable, or expression.

An option that is specified without a value is interpreted as enabled. For example, the following are equivalent:

```
ENVIRONMENT(FIXED_LENGTH_RECORDS)
ENVIRONMENT(FIXED_LENGTH_RECORDS('1'B))
```

For arguments of this type, PL/I converts any non-Boolean value to BIT(1) ALIGNED.

7.1.2 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 apply to the file only if the opening of the file actually results in the creation of a new file. If the file opening causes an existing file to be opened, the option is ignored.

7.1.3 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 for an example of calling it, see Chapter 9.

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

7.1.4 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.

7.1.5 Conflicting and Invalid ENVIRONMENT Options

Conflicting or invalid options or values for options can 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.

7.2 Summary of ENVIRONMENT Options

The options to the PL/I ENVIRONMENT attribute are summarized in alphabetical order in Table 7–1. Columns in Table 7–1 provide the following information:

Option and Usage	Gives the name of the ENVIRONMENT option, its argument (if any), and a brief description of its usage. An option that does not show an argument can be specified with a Boolean argument.

Specify At	Indicates when the option is meaningful. The possible items in this column are as follows: Create—the option can be specified on a DECLARE or OPEN. It is meaningful only when a file is created. Open—the option can be specified on a DECLARE or OPEN. It is meaningful when an existing file is opened. Close—the option can be specified on a DECLARE, OPEN, or CLOSE. It takes effect when the file is closed. Update—the option is meaningful when an existing file is opened with the UPDATE attribute or with the ENVIRONMENT option APPEND.
Valid I/O Types	Indicates whether the option is valid for stream or record files.
Default Value	Indicates the default value, if any, when the option is not specified for a file.
Data Type	Specifies the required data type of the argument.

Appendix B lists the options and their corresponding RMS equivalents.

Table 7–1 Summary of ENVIRONMENT Options

Option and Usage	Specify At	Valid I/O Types	Default Value	Data Type
APPEND Places output for a file at the end of a file.	Create Open	Record Stream	Disabled	BIT(1)
BACKUP_DATE (variable) Overrides the default backup date of the file.	Create Open	Record Stream	Date and time file was last backed up	BIT(64) ALIGNED
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 block 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 BLOCKSIZE Value	FIXED BINARY
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
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)

(continued on next page)

Table 7–1 (Cont.) Summary of ENVIRONMENT Options

Option and Usage	Specify At	Valid I/O Types	Default Value	Data Type
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 the 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)
DEFAULT_FILE_NAME (expression) Defines a default file specification for a file.	Create Open	Record Stream	'.DAT'	CHAR(128)
DEFERRED_WRITE Requests file system optimization of output.	Create Open	Record	Disabled	BIT(1)
DELETE Specifies that the file be deleted when it is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
EXPIRATION_DATE (variable) Defines the expiration date for a magnetic tape file.	Create	Record Stream	Creation date	BIT(64) ALIGNED
EXTENSION_SIZE (expression) Specifies a default extension size for a disk file.	Create Open	Record Stream	System default	FIXED BINARY
FILE_ID (variable) Identifies a file by its internal file identification.	Open	Record Stream	Not applicable	(6)FIXED BINARY
FILE_ID_TO (variable) Identifies a file by its internal file identification.	Create Open	Record Stream	Not applicable	(6)FIXED BINARY
FILE_SIZE (expression) Defines the initial number of blocks to be allocated for a file.	Create	Record Stream	Not applicable	FIXED BINARY
FIXED_CONTROL_SIZE (expression) Defines records as variable length with fixed-length control, and specifies the size of the fixed control area. On open, returns the length of the fixed control area.	Create	Record	Disabled	FIXED BINARY
FIXED_CONTROL_SIZE_TO (variable) Defines records as variable length with fixed-length control and specifies the size of the fixed control area. On open, returns the length of the fixed control area.	Create Open	Record	Disabled	FIXED BINARY
FIXED_LENGTH_RECORDS Specifies a file with fixed-length records of a maximum record size.	Create	Record	Disabled	BIT(1)

(continued on next page)

Table 7–1 (Cont.) Summary of ENVIRONMENT Options

Option and Usage	Specify At	Valid I/O Types	Default Value	Data Type
GROUP_PROTECTION (expression) Defines the type of file access allowed to members of the owner's group.	Create	Record Stream	Current process default	CHAR(4)
IGNORE_LINE_MARKS Specifies that end-of-line characters are not to be treated as field delimiters in GET LIST statements.	Create Open	Stream	Disabled	BIT(1)
INDEX_NUMBER (expression) Specifies the initial index to use in accessing records in an indexed sequential file.	Create Open	Record	0	FIXED BINARY
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
MAXIMUM_RECORD_SIZE (expression) Specifies the maximum size that is valid for any record in the file.	Create	Record	512 bytes[1]	FIXED BINARY
MULTIBLOCK_COUNT (expression) Specifies the number of blocks to be allocated for file system buffering.	Create Open	Record	Current process default	FIXED BINARY
MULTIBUFFER_COUNT (expression) Specifies the number of buffers to be allocated for file system buffering.	Create Open	Record	Current process default	FIXED BINARY
NO_SHARE Prohibits all types of shared access to the file.	Create Open	Record	Enabled[2]	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
OWNER_ID (expression) Specifies the entire 32-bit identifier of the owner of the file; can be used instead of OWNER_GROUP and OWNER_MEMBER.	Create	Record Stream	Current process identifier (UIC)	FIXED BINARY
OWNER_MEMBER (expression) Specifies the member number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process member number	FIXED BINARY

[1]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 48 bytes.

[2]Disabled if the file is opened for input, enabled if opened for output or update.

(continued on next page)

Table 7–1 (Cont.) Summary of ENVIRONMENT Options

Option and Usage	Specify At	Valid I/O Types	Default Value	Data Type
OWNER_PROTECTION (expression) Specifies the type of file access allowed the owner of the file.	Create	Record Stream	Current process default	CHAR(4)
PRINTER_FORMAT Specifies that records in the file will be printed with printer format carriage control embedded in the fixed control area of the records.	Create	Record	Disabled	BIT(1)
READ_AHEAD Requests file system optimization on read operations.	Open	Record Stream	Disabled	BIT(1)
READ_CHECK Requests verification of read operations.	Create Open	Record Stream	Disabled	BIT(1)
RECORD_ID_ACCESS Indicates that records will be accessed by internal file system identification.	Create Open	Record	Disabled	BIT(1)
RETRIEVAL_POINTERS (expression) Specifies the number of file system extent pointers to be maintained for file access.	Create Open	Record Stream	Current system default	FIXED BINARY
REVISION_DATE (variable) Overrides the default revision date of the file.	Close	Record Stream	Date and time file is closed	BIT(64) ALIGNED
REWIND_ON_CLOSE Requests that a magnetic tape volume be rewound when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
REWIND_ON_OPEN Requests that a magnetic tape volume be rewound when the file is opened.	Create Open	Record Stream	Enabled	BIT(1)
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	Enabled[3]	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 (UICs).	Create	Record Stream	Current process default	CHAR(4)

[3] Enabled if the file is opened for input, otherwise disabled.

(continued on next page)

Table 7–1 (Cont.) Summary of ENVIRONMENT Options

Option and Usage	Specify At	Valid I/O Types	Default Value	Data Type
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 the file is closed.	Create Update Close	Record Stream	Disabled	BIT(1)
USER_OPEN (entry-name) Specifies a user-written function to open the file.	Create Open	Record Stream	RMS Open	ENTRY
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)

The following sections describe each option in detail.

7.2.1 APPEND Option

The APPEND option opens an existing file for output so that new records are added following the current end-of-file. The format of this option is as follows:

APPEND [(boolean-expression)]

Rules

- The APPEND option is meaningful only when an existing file is opened with the OUTPUT attribute. It overrides the default action for opening an output file, which is to create a new file with a higher version number.
- If the APPEND option is specified when a file is created, the option is ignored; that is, if the file does not exist, a new file is created.
- APPEND conflicts with the SUPERSEDE option; it is invalid for a file opened with the INPUT or UPDATE attribute.

Usage

Use the APPEND option to open a file and position it at the end-of-file. For example, to add records at the end of a file on a magnetic tape, you can open the file as follows:

```
DECLARE TAPEFILE FILE SEQUENTIAL OUTPUT;
   .
   .
   .
OPEN FILE(TAPEFILE) ENVIRONMENT(APPEND);
```

This OPEN statement opens the file TAPEFILE. The file constant TAPEFILE is assumed to be a logical name. The system translates the logical name to locate the tape device with which it is associated. The tape is positioned at its current end-of-file.

APPEND can also be used for sequential disk files and for mailboxes.

7.2.2 BACKUP_DATE Option

The BACKUP_DATE option lets you specify a date and time field for the file's backup date, allowing you to override the existing backup date and time. The format of this option is as follows:

```
BACKUP_DATE (variable-reference)
```

variable-reference
Specifies the name of a BIT(64) ALIGNED variable containing an absolute time value in system format. The value specifies the date and time to be used as the file's backup date and time.

Rules

The BACKUP_DATE option is meaningful only when the file is opened or created.

Usage

You can obtain the time value required by using the Convert ASCII String to Binary Time system service (SYS$BINTIM). For an example of a call to this procedure to obtain a system time value, see Chapter 11.

7.2.3 BATCH Option

The BATCH option requests that the file be submitted to the system batch job queue when it is closed. The format of this option is as follows:

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

Rules

- The BATCH option can be specified when a file is created, opened, or closed.
- This option applies only to stream files or to sequential record files.
- Once the BATCH option has been specified for a file on a particular file opening, it cannot be disabled.
- BATCH conflicts with the INDEXED option and with the KEYED and DIRECT file description attributes.

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 is marked for deletion after the batch job is completed.

7.2.4 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 as follows:

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

Rules

- The BLOCK_BOUNDARY_FORMAT option is meaningful only when a file is created.
- This option applies only to sequential files; it is ignored if specified for relative or indexed sequential files.
- If the BLOCK_BOUNDARY_FORMAT option is specified for a file, the maximum record size must be less than 512 bytes.

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

7.2.5 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 as follows:

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

Rules

- 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.
- This option applies only to disk files and to magnetic tape files.
- The BLOCK_IO option conflicts with the STREAM file description attribute and with the following ENVIRONMENT options:

  BLOCK_BOUNDARY_FORMAT
  CARRIAGE_RETURN_FORMAT
  FIXED_CONTROL_SIZE
  FIXED_LENGTH_RECORDS
  PRINTER_FORMAT
  RECORD_ID_ACCESS

Usage

Disk Files: When a disk file is opened for block I/O, each READ or WRITE statement always transfers data beginning on a block boundary; multiple physical blocks can be read or written. The number of bytes transferred in an I/O operation depends on the size of the input or output variable specified in the READ or WRITE statement. When a READ statement reads fewer bytes than specified by the size of the input variable, the ERROR condition is signaled; this condition is equivalent to an end-of-file indication.

When a disk file is opened with the BLOCK_IO option and with the KEYED and UPDATE attributes, the file can be accessed with keyed READ, REWRITE, and WRITE statements. In this case, the key value is the virtual block number of a block. The first block is always numbered 1. There is no distinction between the statements REWRITE KEY(n) and WRITE KEYFROM(n); both statements store data in the block numbered n of the file. If the file is a sequential file, it is extended if necessary.

Magnetic Tape Files: In a magnetic tape file, the size of the block is the size specified when the tape was created; if the tape was not previously written, the block size is set when the tape is mounted. Sequential READ and WRITE statements transfer a block at a time.

7.2.6 BLOCK_SIZE Option

The BLOCK_SIZE option specifies the size, in bytes, of the blocks when a magnetic tape file is created. Its format is as follows:

BLOCK_SIZE(integer-expression)

integer-expression
Specifies a numeric value in the range 20 through 65,532, giving the number of bytes in a block for the tape file. If the BLOCK_SIZE option is not specified, or if the expression is specified as 0, the block size specified when the tape volume was mounted is used by default.

Rules

- The BLOCK_SIZE option is meaningful only when a file is created.
- This option applies 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.

7.2.7 BUCKET_SIZE Option

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

BUCKET_SIZE(integer-expression)

integer-expression
Is a fixed binary value in the range 0 through 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

- The BUCKET_SIZE option is meaningful only when a file is created.
- 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 following formulas.

Relative Files with Fixed-Length Records

$$bsiz = ((rlen + 1) * rnum)/512$$

bsiz
Is the number of blocks per bucket, rounded up to the next higher integer. The result must be in the range 1 through 32.

rlen
Is the size of the fixed-length records (specified by the MAXIMUM_RECORD_SIZE option).

rnum
Is the number of records that you want in each bucket.

The overhead required for these files consists of one byte for each record.

Relative Files with Variable-Length Records

$$bsiz = \left(\left(rmax + 3\right) * rnum\right)/512$$

bsiz
Is the number of blocks per bucket, rounded up to the next higher integer. The result must be in the range 1 through 32.

rmax
Is the maximum size of any record in the file (specified by the MAXIMUM_RECORD_SIZE option).

rnum
Is the number of records that you want in each bucket.

The overhead required for these files is three bytes for each record.

Relative Files with Variable Fixed-Length Control Records

$$bsiz = \left(\left(rmax + fsiz + 3\right) * rnum\right)/512$$

bsiz
Is the number of blocks per bucket, rounded up to the next higher integer. The result must be in the range 1 through 32.

rmax
Is the maximum size of the data portion of any record in the file (specified by the MAXIMUM_RECORD_SIZE option).

fsiz
Is the size of the fixed-length control area of records (specified by the FIXED_CONTROL_SIZE option).

rnum
Is the number of records you want in each bucket.

The overhead required for these files consists of three bytes, plus the fixed control size, for each record.

By careful calculation of a bucket size, you can improve I/O 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 *Open VMS Record Management Services Manual*.

7.2.8 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 as follows:

CARRIAGE_RETURN_FORMAT [(boolean-expression)]

Rules

- The CARRIAGE_RETURN_FORMAT option is meaningful only when a file is created.
- 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.

7.2.9 CONTIGUOUS Option

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

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

Rules

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

Usage

By default, a disk file consists of noncontiguous areas, or extents, on a disk volume. 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 I/O operations.

The CONTIGUOUS option is generally used with the FILE_SIZE option to specify exactly how much contiguous space is to be allocated for the file when it is first created. When the FILE_SIZE option is not specified, the size of the first allocation is determined by the default cluster size of the disk (usually three to five blocks).

If there is not enough contiguous space on the given volume for the specified size, 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 RMS status code RMS$_FULL.

If the file need not be entirely contiguous, use the CONTIGUOUS_BEST_TRY option instead of the CONTIGUOUS option.

Note that both the CONTIGUOUS and CONTIGUOUS_BEST_TRY options apply only to the first allocation of space for the file. If the file is later extended in any way, the new space allocations may or may not be contiguous with the first allocation.

7.2.10 CONTIGUOUS_BEST_TRY Option

This option requests that, if possible, disk space for a new file be allocated in contiguous space on the disk. When the file system allocates space, it tries to place the file in contiguous blocks. The format of this option is as follows:

CONTIGUOUS_BEST_TRY [(boolean-expression)]

Rules

- The CONTIGUOUS_BEST_TRY option is meaningful only when a file is created.
- This option applies only to disk files.
- CONTIGUOUS_BEST_TRY overrides the CONTIGUOUS option.

7.2.11 CREATION_DATE Option

The CREATION_DATE option lets you specify a date and time field for the file's creation, allowing you to override the default creation date and time. The format of the CREATION_DATE option is as follows:

CREATION_DATE (variable-reference)

variable-reference

Specifies the name of a BIT(64) ALIGNED variable containing an absolute time value in system format. The value specifies the date and time to be used as the file's creation date and time.

Rules

The CREATION_DATE option is meaningful only when a file is created.

Usage

You can obtain the time value required by using the Convert ASCII String to Binary Time system service (SYS$BINTIM).

7.2.12 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 as follows:

CURRENT_POSITION [(boolean-expression)]

Rules

- The CURRENT_POSITION option is meaningful only when a file is created.
- This option applies only to magnetic tape files.
- 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 magnetic tape and proceed to write another file on the same 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 is executed, the tape identified by the logical name TAPEFILE retains the position it had following the CLOSE statement.

7.2.13 DEFAULT_FILE_NAME Option

The DEFAULT_FILE_NAME option specifies default fields for the file specification associated with the PL/I file reference. Its format is as follows:

```
DEFAULT_FILE_NAME [ (character-expression ) ]
```

character-expression
Is a character-string expression specifying one or more components of an OpenVMS file specification. If no value or a null string is specified, PL/I applies no default values for file specifications.

The string can have a maximum length of 128 characters.

Rules

- The DEFAULT_FILE_NAME option is meaningful when a file is created or opened.
- When the DEFAULT_FILE_NAME option is not specified, PL/I applies the default file type DAT to file specifications that do not contain a file type.

Usage

For an explanation of the steps that PL/I takes to complete a file specification, including its use of the value of the DEFAULT_FILE_NAME option, see Chapter 4.

7.2.14 DEFERRED_WRITE Option

The DEFERRED_WRITE option requests that modified I/O buffers not be written back to the disk file until the buffers are needed for other purposes. The format of this option is as follows:

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

Rules

- The DEFERRED_WRITE option is meaningful when a file is created or opened. An existing file can be opened for update or opened with the APPEND option.
- This option applies only to relative and indexed sequential files.
- If a file is opened with the SHARED_READ or SHARED_WRITE option and the DEFERRED_WRITE option, the DEFERRED_WRITE option will be ignored.

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 9.

7.2.15 DELETE Option

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

DELETE [(boolean-expression)]

Rules

- The DELETE option can be specified when a file is created, opened, or closed.
- 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.

You can also use this option to delete an existing file. For example:

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

When this CLOSE statement is executed, the OpenVMS file associated with the PL/I file constant INFILE is deleted.

7.2.16 EXPIRATION_DATE Option

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

EXPIRATION_DATE (variable-reference)

variable-reference

Specifies the name of a BIT(64) ALIGNED variable that contains an absolute time value or a delta time value in system format. The value specifies the date and time at which a file expires.

Rule

The EXPIRATION_DATE option is meaningful only when a file is created.

Usage

You can obtain the time value required by using the Convert ASCII String to Binary Time system service (SYS$BINTIM).

7.2.17 EXTENSION_SIZE Option

The EXTENSION_SIZE option sets the default extension quantity for a file, that is, the number of blocks to be added to a disk file when a PUT or WRITE operation increases the size of the file beyond its original allocation. The format of the EXTENSION-SIZE option is as follows:

EXTENSION_SIZE(integer-expression)

integer-expression

Is a fixed binary integer in the range 0 through 65,535, indicating the extension quantity in 512-byte blocks.

Rules

- The EXTENSION_SIZE option is meaningful when a file is created or opened. An existing file can be opened for update or for output with the APPEND option.
- This option applies only to disk files.
- When an extension size is specified in the opening of an existing file, the extension value is set for the duration of this file opening. When the file is closed, the default set when the file was created is reestablished.

Usage

Using the EXTENSION_SIZE option can improve the efficiency of I/O operations to files that are frequently enlarged.

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 the 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.

7.2.18 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. The format of the option is as follows:

```
FILE_ID(variable-reference)
```

variable-reference

Specifies the name of a 6-element array variable that gives the file identification obtained when the file was created.

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

Rules

- The FILE_ID option is valid only when an existing file is opened.
- This option conflicts with the TITLE, DEFAULT_FILE_NAME, and FILE_ID_TO options.
- If there is no file with the indicated file identification, the UNDEFINEDFILE condition is signaled.
- The FILE_ID option cannot be used with DECnet.
- This option is provided only for use with the TEMPORARY option.
- You must specify the FILE_ID option to reopen a file that was created with the TEMPORARY option.

7.2.19 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 as follows:

```
FILE_ID_TO(variable-reference)
```

variable-reference
Specifies the name of a 6-element array variable to receive the file identification of the created file.

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

Rules

- The FILE_ID_TO option is meaningful when a file is created or opened.
- This option applies only to disk files.
- FILE_ID_TO conflicts with the FILE_ID option.
- The FILE_ID_TO option cannot be used with DECnet.
- This option is provided only for use with temporary files.

Usage

This option allows you to save the internal file identification of a file created with the TEMPORARY option so that you can access the file later and eventually delete it.

For an example of the FILE_ID_TO and FILE_ID options used for temporary files, see the following description of the TEMPORARY option.

7.2.20 FILE_SIZE Option

The FILE_SIZE option lets you specify the number of blocks to be initially allocated for a file. The format of the FILE_SIZE option is as follows:

```
FILE_SIZE(integer-expression)
```

integer-expression
Is a value in the range 0 through 4,294,967,295, giving the number of 512-byte blocks. A value of 0 indicates no allocation.

On OpenVMS VAX. 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 for OpenVMS VAX), you must express the number as a negative value; RMS interprets the number as an unsigned integer.

Rules

- The FILE_SIZE option is meaningful only when a file is created.
- This option applies only to disk files.

Usage

The FILE_SIZE option can optimize I/O operations on large files. When you initially create a file that will require a large amount of space and to which new records will be added frequently, you can reduce the file system overhead required to allocate space each time the file is extended by requesting an initial allocation amount. For example:

```
DECLARE MONTHLY_TRANSACT FILE RECORD OUTPUT
        ENVIRONMENT (FILE_SIZE (128));
```

If you do not specify the FILE_SIZE option, or if you specify a file size of 0, PL/I uses the default extension quantity for the file when the first write or put operation occurs on the file. The default extension quantity is defined in the EXTENSION_SIZE option or supplied by default.

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.

If you allocate more space for a file than it requires, the unused space is wasted.

7.2.21 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 as follows:

FIXED_CONTROL_SIZE(integer-expression)

integer-expression
Is an integer expression in the range 0 through 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

- The FIXED_CONTROL_SIZE option is meaningful only when a file is created.
- This option applies only to relative and sequential files with variable-length records.
- The FIXED_CONTROL_SIZE option conflicts with the BLOCK_IO and INDEXED options and with the STREAM and UPDATE file description attributes.
- 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 can 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, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP write zeros in the fixed control area of the record.

7.2.22 FIXED_CONTROL_SIZE_TO Option

When the FIXED_CONTROL_SIZE_TO option is used to open an existing file with fixed control records, PL/I returns the length of the fixed control area to a user-specified variable. The format of this option is as follows:

FIXED_CONTROL_SIZE_TO(variable-reference)

variable-reference
Specifies the name of a fixed binary variable to receive the length of the fixed control area.

The variable must be declared as FIXED BINARY.

Rules

- The FIXED_CONTROL_SIZE_TO option is valid only when an existing file is opened.
- This option applies only to relative and sequential files with variable-length records.
- If this option is specified for a file that does not have fixed control records, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP return a zero to the specified variable.

7.2.23 FIXED_LENGTH_RECORDS Option

The FIXED_LENGTH_RECORDS option specifies that all records in the file are to be of the same length. If you do not specify this option when you create a file, the records in the file will be variable length by default. The format of this option is as follows:

FIXED_LENGTH_RECORDS [(boolean-expression)]

Rules

- The FIXED_LENGTH_RECORDS option is meaningful only when a file is created.
- The FIXED_LENGTH_RECORDS option conflicts with the FIXED_CONTROL_SIZE and BLOCK_IO options and with the combination of the STREAM and OUTPUT file description attributes.

Usage

When the FIXED_LENGTH_RECORDS option is specified for the creation of a file, the size of each record can be specified with the MAXIMUM_RECORD_SIZE option. If MAXIMUM_RECORD_SIZE is not specified, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP provide a default length of 512 bytes for sequential files and a default length of 480 bytes for relative files.

7.2.24 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 as follows:

GROUP_PROTECTION (character-expression)

character-expression

Is a 1- to 4-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 can be repeated, but the maximum length of the string is 4 characters. All other characters are

invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

Rules

- The GROUP_PROTECTION option is meaningful only when a file is created.
- 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.

7.2.25 IGNORE_LINE_MARKS Option

The IGNORE_LINE_MARKS option overrides the default manner in which VAX 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 as follows:

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

Rules

- The IGNORE_LINE_MARKS option can be specified when a file is opened.
- 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.

Usage

When IGNORE_LINE_MARKS is specified for a stream file, an end-of-line terminator such as a carriage return or escape character is ignored, and a GET statement continues to read characters in the input stream until a space, tab, or comma is encountered. For example:

```
123 Return
456,
```

If these lines are entered in response to a GET statement, the resulting input variable is given the value 123456; the carriage return is ignored, and the GET statement reads input data until the comma is encountered.

7.2.26 INDEX_NUMBER Option

The INDEX_NUMBER option specifies which index is to be used initially to process an indexed sequential file. Its format is as follows:

```
INDEX_NUMBER(integer-expression)
```

integer-expression

Specifies the initial index to be used. The value of the expression must be the number of an index for records in the file, where the primary index is 0, the secondary index is 1, and so on.

Rules

- The INDEX_NUMBER option can be specified when a file is opened for input or update.
- This option is meaningful only for indexed sequential files.

Usage

If the INDEX_NUMBER option is specified for opening an indexed sequential file for sequential access, the records will be accessed in the order of their occurrence in the index specified by the INDEX_NUMBER option.

If the file is opened for keyed access, the INDEX_NUMBER option specifies the initial index to be used. This value can be changed in the following statements:

- Any subsequent record I/O statement that specifies the INDEX_NUMBER option
- A WRITE statement that specifies the KEYFROM option
- Any statement that specifies the RECORD_ID option

In the latter two cases, the index number is set to 0, the primary index.

7.2.27 INDEXED Option

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

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

Rules

- The INDEXED option is meaningful when an existing file is opened.
- This option applies only to indexed sequential files.
- 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 can 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.

7.2.28 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.

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.

The format of this option is as follows:.

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

Rules

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

7.2.29 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 as follows:

```
MAXIMUM_RECORD_NUMBER(integer-expression)
```

integer-expression
For OpenVMS VAX, a numeric expression that must yield an integer result in the range 0 through 2,147,483,647. For OpenVMS AXP, ??.

If you specify 0, or if this option is not specified, there is no maximum number and no run-time checking of record numbers is performed.

Rules

- The MAXIMUM_RECORD_NUMBER option is meaningful only when a file is created.
- This option applies only to relative files.
- MAXIMUM_RECORD_NUMBER conflicts with the INDEXED option and with the STREAM file description attribute.

Usage
The MAXIMUM_RECORD_NUMBER option lets you specify an upper limit to the values that can be specified for relative record numbers in the file. When a maximum number has been set, then the file system checks the relative number of each record that is written to the file. If a relative record number is not in the correct range, the KEY condition is signaled. If referenced in an ON-unit for this condition, the ONCODE built-in function returns the value associated with the RMS status code RMS$_MRN.

7.2.30 MAXIMUM_RECORD_SIZE Option

The MAXIMUM_RECORD_SIZE option specifies the largest size that records in a file can have. The actual meaning of this option varies according to the type of file:

- For a file with fixed-length records, the maximum record size indicates the size of each record in bytes.
- For a file with variable-length records, the maximum record size is the size in bytes of the largest record that can be written to the file.
- For variable-length records with fixed-length control, the maximum record size does not include the fixed control area size.
- For relative files, the maximum record size is used in conjunction with the bucket size to determine the cell size in bytes.

The format of this option is as follows:

```
MAXIMUM_RECORD_SIZE(integer-expression)
```

integer-expression
Is a numeric expression with values in the range 1 to a maximum determined by record format and file organization, as in the following table.

File Organization	Record Format	Maximum
Sequential	Fixed or variable length	32,767
Relative	Fixed length	31,998
Relative	Variable length	31,998
Indexed sequential	Fixed length	16,362

File Organization	Record Format	Maximum
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.

7.2.31 MULTIBLOCK_COUNT Option

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

```
MULTIBLOCK_COUNT(integer-expression)
```

integer-expression

Is a fixed binary expression in the range 0 through 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

- The MULTIBLOCK_COUNT option is meaningful when a file is created or opened.
- This option applies only to sequential disk files. It is ignored if the BLOCK_IO option is specified or if the file is not a sequential disk file.

Usage

The MULTIBLOCK_COUNT option can optimize I/O operations on sequential disk files. By default, RMS transfers data in 512-byte disk blocks. To improve I/O access time, you can specify a multiple of 512-byte blocks to specify that a larger number of blocks be transferred with each input or output operation. In general, a multiblock count of between 12 and 16 results in good performance for sequential I/O.

The MULTIBLOCK_COUNT option can also be used with the MULTIBUFFER_COUNT option to request a specified number of I/O buffers, each of which can contain the given number of blocks.

7.2.32 MULTIBUFFER_COUNT Option

The MULTIBUFFER_COUNT option specifies the number of buffers to be allocated for file operations; it has the following effects, depending on the organization of the file:

- For relative and indexed sequential files, it results in a cache of buckets that can improve random access.
- For sequential files, multiple buffers allow throughput to be increased on file transfers when either the READ_AHEAD or the WRITE_BEHIND option is also selected. If neither of these options is specified, the MULTIBUFFER_ COUNT option is meaningless for sequential files.

Its format is as follows:

```
MULTIBUFFER_COUNT(integer-expression)
```

integer-expression
Specifies a value in the range –128 through 127, indicating the number of buffers to be allocated; RMS uses the absolute value of the field. If 0 is 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.

If either the READ_AHEAD or the WRITE_BEHIND option is specified and the MULTIBUFFER_COUNT option is not specified, PL/I uses the RMS default value of two buffers.

Rules

- The MULTIBUFFER_COUNT option is meaningful when a file is created or opened.
- This option applies only to disk files.
- 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 the following ENVIRONMENT specification:

```
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.

7.2.33 NO_SHARE Option

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

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

Rules

- The NO_SHARE option is meaningful when a file is created or opened.
- This option applies only to disk files.
- NO_SHARE conflicts with the SHARED_READ and SHARED_WRITE options.
- If the specified file is already opened for sharing, the UNDEFINEDFILE condition is signaled.

Usage

By default, the NO_SHARE option is enabled when a file is opened for output or update. This option is disabled when a file is opened for input.

7.2.34 OWNER_GROUP Option

The OWNER_GROUP option overrides the default group number in the user identification code (UIC) associated with the file's owner. The group number, together with the member number, defines the ownership of the file and provides the values against which protection is applied. Its format is as follows:

```
OWNER_GROUP(integer-expression)
```

integer-expression

Is an integer value in the range 0 through 16383 (decimal).

Rules

- The OWNER_GROUP option is meaningful only when a file is created.
- If the OWNER_GROUP option is not specified, PL/I uses the group number in the current UIC.
- The OWNER_GROUP option conflicts with the OWNER_ID option.
- To specify an owner UIC for a file that is different from the UIC under which the current program is executing, the process must have the SYSPRV user privilege or a system UIC.

Usage

Note that although the value can be specified to PL/I in decimal, the OpenVMS system always displays UICs in octal format.

Following is an example of a program using the OWNER_GROUP and the OWNER_MEMBER options to define the ownership of a file. Note the use of the DECODE built-in function to obtain octal numbers for the UIC.

```
OWNER: PROCEDURE OPTIONS(MAIN);

    DCL TEST_FILE STREAM OUTPUT FILE;

    OPEN FILE(TEST_FILE)
        ENVIRONMENT(OWNER_GROUP(DECODE('214',8)),
            OWNER_MEMBER(DECODE('10223',8)));

    PUT FILE(TEST_FILE) LIST('This file is owned by UIC [214,10223].');

    CLOSE FILE(TEST_FILE);

    END OWNER;
```

7.2.35 OWNER_ID Option

The OWNER_ID option overrides the default owner of a file. This option combines the capabilities of the OWNER_GROUP and OWNER_MEMBER options. In addition, this option allows the specification of identifiers that are not true UICs, which is not possible with the other two options. Its format is as follows:

OWNER_ID(integer-expression)

integer-expression
Is any FIXED BINARY value.

Rules

- The OWNER_ID option is meaningful only when a file is created.
- If the OWNER_ID option is not specified, PL/I uses the current process UIC.
- This option conflicts with both the OWNER_GROUP and the OWNER_MEMBER options.

Usage

Following is an example of a program using the OWNER_ID option:

```
/*
 *  This program example creates a file owned by the identifier TEST.  (Note
 *  that the process running this program must hold the identifier with the
 *  RESOURCE attribute or the file creation will fail.)
 */
%REPLACE IDENTIFIER BY 'TEST';

OWNER_ID: PROCEDURE OPTIONS(MAIN);

    %INCLUDE $KGBDEF;
    %INCLUDE $STSDEF;
    %INCLUDE SYS$ASCTOID;

    DCL TEST_FILE STREAM OUTPUT FILE;
    DCL IDENTIFIER_VALUE FIXED BINARY;
    DCL KGB_BITS BIT(32) ALIGNED;

    /*
     *  Get the value of the identifier.  (Note: The fact that the value
     *  translates successfully does not mean that the current process
     *  holds the identifier.
     */
    STS$VALUE = SYS$ASCTOID( IDENTIFIER, IDENTIFIER_VALUE, KGB_BITS );
    IF ~STS$SUCCESS THEN SIGNAL VAXCONDITION(STS$VALUE);
```

```
/*
 *  Does the identifier have the resource attribute?  (Note: If you
 *  want to determine whether this process holds the identifier with the
 *  resource attribute, you must use the SYS$FIND_HOLDER or SYS$CHKPRO
 *  system services.)
 */
IF (KGB_BITS & KGB$M_RESOURCE) = '0'b
THEN
    PUT LIST('The identifier ' ||
        IDENTIFIER || ' does not have the RESOURCE attribute.');

OPEN FILE(TEST_FILE) ENVIRONMENT(OWNER_ID(IDENTIFIER_VALUE));

PUT FILE(TEST_FILE)
    LIST('This file is owned by the identifier ' || IDENTIFIER || '.');

CLOSE FILE(TEST_FILE);

END OWNER_ID;
```

7.2.36 OWNER_MEMBER Option

The OWNER_MEMBER option overrides the default member number in the user identification code (UIC) 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 as follows:

OWNER_MEMBER(integer-expression)

integer-expression
Is a numeric value in the range 0 through 65535 (decimal).

Rules

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

Usage
Note that although the value can be specified to PL/I in decimal, the OpenVMS system always displays UICs in octal format.

See the section on the OWNER_GROUP option for an example of a program using the OWNER_MEMBER option.

7.2.37 OWNER_PROTECTION Option

The OWNER_PROTECTION option defines the type of access to be permitted to the file by the file's owner and by other users with the same user identification code (UIC). The format of this option is as follows:

OWNER_PROTECTION(character-expression)

character-expression
Is a 1- to 4-character string expression indicating the access privileges to be granted to the file's owner (and any other users who have the same UIC). 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 can be repeated, but the maximum length of the string is 4 characters. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

Rules

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

7.2.38 PRINTER_FORMAT Option

The PRINTER_FORMAT option specifies that the records in the file contain printing and carriage-control information in the fixed control area. The format of this option is as follows:

PRINTER_FORMAT [(boolean-expression)]

Rules

- The PRINTER_FORMAT option is meaningful only when a file is created.
- 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.
- This option applies only to relative or sequential files.
- 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, 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.

Note that format items are implemented in the following way:

- Pages are started with a form feed.
- Blank lines (created with the SKIP option) are totally blank.
- Tabs are counted as spaces.

Table 7–2 summarizes 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 determines the carriage control to be performed before writing the record. The second byte is the *postfix byte*: it determines the carriage control to be performed after writing the record. The values shown in Table 7–2 have the same meanings in either byte; the bytes are interpreted separately.

Table 7–2 Printer File Format Carriage Control

Bit 7	Bits 0–6	Meaning
0	0	No carriage control is specified, that is, NULL.
0	1–7F	Bits 0 through 6 are a count of new lines (line feeds followed by carriage return).

Bit 7	Bit 6	Bit 5	Bits 0–4	Meaning
1	0	0	0–1F	Output the ASCII control character specified by the configuration of bits 0 through 4 (7-bit character set).
1	1	0	0–1F	Output the ASCII control character specified by the configuration of bits 0 through 4, which are translated as ASCII characters 128 through 159 (8-bit character set).
1	1	1	0–1F	Reserved.

Example 7–1 illustrates a procedure that uses explicit carriage control. The following notes are keyed to this program:

1 The structures LINE_FEEDS and CARRIAGE_CONTROL define bit fields corresponding to the fields shown in Table 7–2. The fields that must be set are initialized.

2 The procedure declares and defines values for NEW_LINE and NEW_PAGE. These 8-bit strings correspond to the fields within the structure CARRIAGE_CONTROL.

Note the use of the POSINT built-in function to define these fields. The POSINT built-in function must be used so that the bit string value is treated as a positive integer.

3 The output file PRINTFILE is declared with the RECORD and OUTPUT attributes and with ENVIRONMENT options FIXED_CONTROL_SIZE and PRINTER_FORMAT.

4 The first line is output with no carriage control. Note that any subsequent lines printed without carriage control would result in overprinting.

5 The procedure uses the POSINT built-in function to specify an integer value for the 7-bit field LINE_FEEDS.COUNT. Then, the variable CONTROL_FIELD is assigned two 1-byte values:

- The prefix byte specifies the number of line feeds to precede the line when it is output. This value is specified by using the STRING built-in function to concatenate the bit fields in LINE_FEEDS.
- The postfix byte specifies a carriage return following the output record. This value is specified using the variable NEW_LINE.

6 The variable CONTROL_FIELD is assigned new values for the prefix and postfix bytes to output another record. This prefix byte specifies a new page; the postfix byte specifies a new line.

7 The line is output.

8 The file is spooled to the system printer when it is closed.

Example 7–1 Explicit Carriage Control

```
PRINTER_FORMAT_EXAMPLE: PROCEDURE OPTIONS(MAIN);

    /*
     * Declare structure definitions for carriage-control bit fields
     * and a FIXED BIN(15) variable for the fixed control area
     */                                                    /* 1  */
    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 carriage returns
     */
    DECLARE
        (NEW_LINE,NEW_PAGE) BIT(8); /* 2  */

    /*
     * Declare PRINTFILE with character-string variable for I/O
     */
    DECLARE
        PRINTFILE RECORD OUTPUT FILE ENV(
                FIXED_CONTROL_SIZE(2), /* 3  */
                PRINTER_FORMAT),
        PRINTREC CHARACTER(80) VARYING;

    /*
     * Set up the NEW_PAGE and NEW_LINE variables using the
     * CARRIAGE_CONTROL structure as a template.
     */
    POSINT(CODE) = 12;            /* Assign ASCII code for form feed */
    NEW_PAGE = STRING(CARRIAGE_CONTROL);

    POSINT(CODE) = 13;            /* Assign ASCII code for CR */
    NEW_LINE = STRING(CARRIAGE_CONTROL);

    OPEN FILE(PRINTFILE);

    /*
     * Output first line with no carriage control
     */
    PRINTREC = 'Output first line with no carriage control'; /* 4  */
    WRITE FILE(PRINTFILE) FROM(PRINTREC);

    /*
     * Prepare to output five line feeds followed by a new line
     */
    POSINT(LINE_FEEDS.COUNT) = 5;  /* assign 5 to LINE_FEEDS.COUNT */ /* 5  */
    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; /* 6  */
    PRINTREC = 'New page';
```

(continued on next page)

Example 7–1 (Cont.) Explicit Carriage Control

```
WRITE FILE(PRINTFILE) FROM(PRINTREC) OPTIONS( /* 7  */
           FIXED_CONTROL_FROM(CONTROL_FIELD));

CLOSE FILE(PRINTFILE) ENV(SPOOL); /* 8  */

END PRINTER_FORMAT_EXAMPLE;
```

7.2.39 READ_AHEAD Option

The READ_AHEAD option requests the overlapping of reading records into buffers with computing operations. This option, used in conjunction with the MULTIBUFFER_COUNT option, can increase the efficiency of I/O operations to disk files. PL/I for OpenVMS VAX and PL/I for OpenVMS AXP enable this option by default. The format of this option is as follows:

READ_AHEAD [(boolean-expression)]

Rules

- The READ_AHEAD option is meaningful when an existing file is opened for input or for update.
- This option applies only to sequential disk files.

Usage

When you use the READ_AHEAD option, you can specify the number of buffers to be used in the MULTIBUFFER_COUNT option. When READ_AHEAD is in effect and no multibuffer count is specified, RMS uses two buffers by default.

When READ_AHEAD is enabled, the data transfer and the reading ahead are transparent to the PL/I program.

7.2.40 READ_CHECK Option

The READ_CHECK option specifies that all input transfers of data between a program and a disk device be followed by a comparison operation to ensure that the data was transferred intact. The format of this option is as follows:

READ_CHECK [(boolean-expression)]

Rules

The READ_CHECK option is meaningful when a file is created or opened. An existing file can be opened for input or for update.

Usage

This option is useful for applications that must verify all I/O operations to ensure that data was successfully transferred. However, use of this option decreases the speed and efficiency of I/O operations.

If READ_CHECK is specified when a file is created, READ_CHECK is the default for all subsequent openings of the file, unless explicitly disabled.

7.2.41 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 as follows:

RECORD_ID_ACCESS [(boolean-expression)]

Rules

- The RECORD_ID_ACCESS option is meaningful when a file is created or opened.
- This option applies only to disk files.
- 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 8.

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 opening. However, a statement cannot specify a record both by key and by record identification.

7.2.42 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 as follows:

```
RETRIEVAL_POINTERS(integer-expression)
```

integer-expression

Is a fixed binary expression in the range –1 through 127. A value in the range of 1 through 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

When a disk is initialized, the default window size is set by the /WINDOW qualifier of the DCL command INITIALIZE. You can override this value for the opening of a specific file by specifying the RETRIEVAL_POINTERS option to increase the speed with which records can be accessed.

However, you should avoid specifying a value that is too large. Space for the pointers is allocated from system space, and a large number of pointers could have an adverse effect on system performance.

7.2.43 REVISION_DATE Option

The REVISION_DATE option lets you specify a date and time field for the file's revision, allowing you to override the default revision date and time. The format of this option is as follows:

```
REVISION_DATE (variable-reference)
```

variable-reference
Specifies the name of a BIT(64) ALIGNED variable containing an absolute time value in system format. The value specifies the date and time to be used as the file's revision date and time.

Rules
The REVISION_DATE option is meaningful only when the file is closed.

Usage
You can obtain the time value required by using the Convert ASCII String to Binary Time system service (SYS$BINTIM).

7.2.44 REWIND_ON_CLOSE Option

The REWIND_ON_CLOSE option specifies, for a file on a magnetic tape volume, that the volume is to be rewound when the file is closed. The format of this option is as follows:

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

Rules

- The REWIND_ON_CLOSE option can be specified when a file is created, opened, or closed.
- This option applies only to magnetic tape files.
- REWIND_ON_CLOSE takes precedence over the CURRENT_POSITION option.

7.2.45 REWIND_ON_OPEN Option

The REWIND_ON_OPEN option specifies, for a file on a magnetic tape volume, that the volume is to be rewound when the file is created or opened. The format of this option is as follows:

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

Rules

- The REWIND_ON_OPEN option is meaningful when a file is created or opened.
- This option applies only to magnetic tape files.
- REWIND_ON_OPEN takes precedence over the CURRENT_POSITION option.

Usage
Magnetic tape file positioning is described in Chapter 6.

7.2.46 SCALARVARYING Option

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

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

Rules

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

Usage

The SCALARVARYING option has the following effect on I/O 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 2-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 as follows:

- 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.

The following example illustrates reading a file with the SCALARVARYING option (presumably the file was created with the SCALARVARYING option in effect):

```
DECLARE EOF BIT(1) ALIGNED INITIAL('0'B);
DECLARE STRING CHARACTER(80) VARYING,
        INFILE FILE RECORD INPUT;

OPEN FILE(INFILE) ENVIRONMENT(SCALARVARYING);
ON ENDFILE(INFILE) EOF = '1'B;
READ FILE(INFILE) INTO(STRING);
DO WHILE (^EOF);
       PUT SKIP LIST(LENGTH(STRING),STRING);
       READ FILE(INFILE) INTO(STRING);
       END;
```

The file defined as INFILE must have 82-byte records: the first two bytes of each record must contain the length of the data within the record. This READ statement reads 82 bytes, and uses the first two as the length and contents of each string.

7.2.47 SHARED_READ Option

The SHARED_READ option specifies that other users who have concurrent access to the file can read records in it. The format of this option is as follows:

SHARED_READ [(boolean-expression)]

Rules

- The SHARED_READ option is meaningful when a file is created or opened.
- This option applies to relative and indexed sequential files.
- SHARED_READ conflicts with the NO_SHARE option.

Usage

By default, the SHARED_READ option is disabled when a file is opened for output or update; that is, sharing is not allowed by default if anyone is writing to the file. SHARED_READ is enabled when a file is opened for input, that is, sharing is allowed if no one is writing to the file.

7.2.48 SHARED_WRITE Option

The SHARED_WRITE option specifies that other users who have concurrent access to the file can write, update, and delete records in the file. The format of this option is as follows:

SHARED_WRITE [(boolean-expression)]

Rules

- The SHARED_WRITE option is meaningful when a file is created or opened.
- This option applies to sequential, relative, and indexed sequential files.
- SHARED_WRITE conflicts with the NO_SHARE option.
- 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.

7.2.49 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 as follows:

SPOOL [(boolean-expression)]

Rules

- The SPOOL option can be specified when a file is created, opened, or closed.
- This option applies to stream files as well as to record files of any file organization.
- Once the SPOOL option has been specified for a particular file and opening, 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 LPC0: (the printer device).

7.2.50 SUPERSEDE Option

The SUPERSEDE option specifies that if a file already exists with the same name, type, and version number as the file specified, the existing file should be replaced. The format of this option is as follows:

SUPERSEDE [(boolean-expression)]

Rules

- The SUPERSEDE option is meaningful only when a file is created.
- SUPERSEDE conflicts with the APPEND option.

Usage

By default, the file system creates a new file and assigns it a new version number whenever a file is opened for output. Consequently, if a file specification does not include a version number, many versions of a file may be created. If the file's TITLE option or DEFAULT_FILE_NAME option specifies an explicit version number, the ERROR condition is signaled if a file with that version number already exists.

In some cases, you may want to specify an explicit version number to ensure that a single version of a specific file be maintained. In these cases, specify the SUPERSEDE option in conjunction with a TITLE value or DEFAULT_FILE_NAME value to ensure that multiple versions of the file are not created.

7.2.51 SYSTEM_PROTECTION Option

The SYSTEM_PROTECTION option defines the type of access to be permitted to the file by users with system user identification codes (UICs). The format of this option is as follows:

SYSTEM_PROTECTION(character-expression)

character-expression

Is a 1- to 4-character string expression indicating the access privileges to be granted to users with system UICs or with the SYSPRV user privilege. 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 can be repeated, but the maximum length of the string is 4 characters. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

Rules

- The SYSTEM_PROTECTION option is meaningful only when a file is created.
- 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.

7.2.52 TEMPORARY Option

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

TEMPORARY [(boolean-expression)]

Rules

- The TEMPORARY option is meaningful only when a file is created.
- 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 can be deleted when it is closed or, if needed later, can be deleted after it has been reused. You specify deletion by using the DELETE option when you open, reopen, or close the file.

However, because no directory entry is created for a temporary file, the file can be reaccessed only by its internal file identification. You can obtain this identification by specifying the FILE_ID_TO option when the file is created. For example:

```
DECLARE WORKFILE FILE OUTPUT SEQUENTIAL,
        WORKFILE_ID (6) FIXED BINARY;

OPEN FILE(WORKFILE) ENVIRONMENT (
                        TEMPORARY,
                        FILE_ID_TO(WORKFILE_ID));
   .
   .
   .
CLOSE FILE(WORKFILE);
   .
   .
   .
OPEN FILE (WORKFILE) ENVIRONMENT (
                              FILE_ID(WORKFILE_ID),
                              DELETE);
```

These statements declare the file WORKFILE, open it with the FILE_ID_TO option, close it, and later reopen it, using the FILE_ID option and specifying the file identification obtained when the file was first opened. The second OPEN statement also specifies the DELETE option of ENVIRONMENT, so that the file is deleted when it is subsequently closed.

Note that the FILE_ID and the FILE_ID_TO options, which are necessary for reaccessing a temporary file, cannot be used across the DECnet.

The TEMPORARY option is also useful in conjunction with the BATCH or SPOOL options. For example, if you create a file that is to be printed but that can be deleted after printing, you can specify it as follows:

```
DECLARE PRINTFILE FILE  PRINT ENVIRONMENT (
                    SPOOL, TEMPORARY, DELETE);
```

When this file is closed, it is automatically queued for printing. Once it is printed, it is deleted.

7.2.53 TRUNCATE Option

The TRUNCATE option specifies that any unused space allocated for a file be deallocated when the file is closed. The file is truncated to its logical end-of-file. The format of this option is as follows:

TRUNCATE [(boolean-expression)]

Rules

- The TRUNCATE option can be specified when a file is created, opened, or closed. An existing file can be opened for update or opened with the APPEND option.
- This option applies only to sequential files.
- Once the TRUNCATE option has been specified for a file on a particular open, it cannot be disabled.

Usage

You can specify this option to conserve disk space. If a file's allocation is greater than its contents require, 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.

7.2.54 USER_OPEN Option

The USER_OPEN option allows you to access RMS facilities not explicitly available in PL/I for OpenVMS VAX and PL/I for OpenVMS AXP by writing a function that controls the opening of the file. Specifying the USER_OPEN option causes the run-time library to call your function to open the file instead of calling RMS to open it according to its normal defaults. The format of this option is as follows:

USER_OPEN (entry-name)

entry-name

An entry variable or entry constant.

When the OPEN statement is executed, the run-time library sets up the RMS file access block (FAB) and the record access block (RAB), as well as its own internal data structures. These blocks transmit requests for file and record operations to RMS; they also return the data contents of files, information about file characteristics, and status codes. For more information on the RAB and the FAB, see the *OpenVMS Record Management Services Reference Manual*.

In order, the three parameters passed to the user-open function by the run-time library are as follows:

- FAB address
- RAB address

- Open-flag, which is passed as a longword 1 if the file exists (in which case SYS$OPEN should be called); if the file does not exist, a longword zero is passed (in which case SYS$CREATE should be called)

Rules

- The function must call SYS$OPEN or SYS$CREATE.
- The status of the call must be returned.
- The function can modify the FAB or the RAB, or both.
- The function can store FAB and RAB values in program variables.
- The structures for the FAB and the RAB that are found in PLI$STARLET (modules $FABDEF and $RABDEF) should be used when you manipulate the FAB and the RAB.
- The USER_OPEN option should be used only when there is no way to do what you want within PL/I.

Note

Your user-open function may have to be changed when new run-time libraries are released.

Usage

The following example shows a PL/I for OpenVMS VAX program that creates a file 1000 blocks long.

```
/*
 * This program allocates 1000 blocks to the file new.tmp.
 */
OPEN_TEST: PROC OPTIONS(MAIN);

DCL F FILE OUTPUT;

OPEN FILE(F) ENVIRONMENT(USER_OPEN(MY_OPEN))  TITLE('NEW.TMP');

RETURN;

    /*
     * This function sets the appropriate bit in the FAB to allocate
     * 1000 blocks for the file.
     */
    MY_OPEN:PROC(FAB,RAB,OPEN_FLAG) RETURNS(FIXED BIN);

    %INCLUDE $FABDEF;
    %INCLUDE $RABDEF;

    DCL 1 FAB LIKE FABDEF;
    DCL 1 RAB LIKE RABDEF;
    DCL OPEN_FLAG FIXED BIN;
    DCL STATUS FIXED BIN;
    %INCLUDE SYS$OPEN;
    %INCLUDE SYS$CREATE;
```

```
    /*
     * Store the allocation quantity.
     */
    FAB.FAB$L_ALQ = 1000;
    /*
     * Call sys$open or sys$create and return its status to the Run-Time
     * Library.
     */
    IF OPEN_FLAG = 1 THEN
        STATUS = SYS$OPEN(FAB,,);
    ELSE
        STATUS = SYS$CREATE(FAB,,,);
    RETURN(STATUS);

    END MY_OPEN;

END OPEN_TEST;
```

7.2.55 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 as follows:

WORLD_PROTECTION(character-expression)

character-expression
Is a 1- to 4-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 can be repeated, but the maximum length of the string is four characters. All other characters are invalid. If any other character is present in the string, the UNDEFINEDFILE condition is signaled.

Rules

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

7.2.56 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 as follows:

WRITE_BEHIND [(boolean-expression)]

Rules

- The WRITE_BEHIND option is meaningful when a file is created or opened. An existing file can be opened either for update or for output with the APPEND option.
- This option applies only to sequential files; it is ignored for unit record devices.

Usage

When you use the WRITE_BEHIND option, you can specify the number of buffers to be used in the MULTIBUFFER_COUNT option. If you specify WRITE_BEHIND and do not specify a multibuffer count, RMS uses two buffers by default.

When the WRITE_BEHIND option is in effect, there is no way for the program to determine when a buffer has been written to disk. To ensure the integrity of a file that is being processed with the WRITE_BEHIND option, you can use the FLUSH built-in subroutine to periodically write all buffers back to disk. The FLUSH built-in subroutine is described in Chapter 9.

7.2.57 WRITE_CHECK Option

The WRITE_CHECK option specifies that all write transfers of data between a program and a disk device be followed by a compare operation to ensure that the data was transferred intact. The format of this option is as follows:

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

Rules

The WRITE_CHECK option is meaningful when a file is created or opened. An existing file can be opened either for update or for output with the APPEND option.

Usage

This option is useful for applications that must verify all I/O operations, to ensure that data was successfully transferred. However, use of this option decreases the speed and efficiency of I/O operations.

If WRITE_CHECK is specified when a file is created, WRITE_CHECK is the default for all subsequent openings of the file, unless explicitly disabled.

7.3 ENVIRONMENT Options for File Protection and File Sharing

This section discusses the ENVIRONMENT options that take advantage of special RMS processing options for file protection and file sharing.

7.3.1 File Protection

Each user who is authorized to use the system is assigned a user identification code (UIC) 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 which other users on the system can access the file and 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.

The types of access privileges defined for a file are as follows:

- Read access—the right or permission to perform an input operation
- Write access—the right or permission to perform an output or update operation
- Execute access—the right or permission to execute an image file
- Delete access—the right or permission to delete the file

You can explicitly control the protection applied to a file in two ways:

- Specify the file's ownership.
- Specify the type of access allowed each category of user.

In a PL/I program, you can specify a file's ownership and protection when you create the file.

7.3.1.1 Defining a File's Ownership

When you specify the ENVIRONMENT attribute for a file you are creating in a PL/I program, you can specify the following options to define the owner of the file, overriding the default:

OWNER_MEMBER
OWNER_GROUP

These options specify the member number and group number of the owner of the file. Specify values for these options using fixed binary expressions. For example:

```
ENVIRONMENT (
            OWNER_GROUP (240),
            OWNER_MEMBER (5))
```

This example defines the owner of the file as any process that has the UIC [360,5]. Note that although the value can be specified to PL/I in decimal radix, the OpenVMS system always displays and interprets UICs in octal radix.

To specify an owner UIC for a file that is different from the UIC under which the current program is executing, the process must have the SYSPRV user privilege or a system UIC.

7.3.1.2 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—for files containing executable program images, gives 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 unspecified categories 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 *OpenVMS DCL Dictionary*.

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.

7.3.2 File Sharing

RMS allows multiple programs to access records in the same file concurrently. The type of access is controlled by ENVIRONMENT specifications and by the current status of the file, that is, whether the file is open and, if it is open, whether it is open for input, output, or update.

The rules for sharing are as follows:

- Sharing is valid only for disk files.
- Sequential, relative, and indexed files can be read-shared; that is, any number of programs can read records in these files at the same time.
- Relative and indexed sequential files can be write-shared; that is, any number of programs can read and write records in these files at the same time.

- Sequential disk files can be write-shared only if there is a single program writing them at a time. Only one program can be writing a sequential file while other programs are reading it.

When you write a PL/I program or programs that will be sharing a file, you can specify the type of sharing. During execution of programs that share a file, RMS ensures the following:

- If a file is already opened when another program attempts to open it, the file is available for sharing.
- Only one program is writing a record at one time.

7.3.2.1 Specifying File Sharing

In a PL/I program, you can specify one of the following file-sharing options in the ENVIRONMENT attribute:

NO_SHARE
SHARED_READ
SHARED_WRITE

These options indicate the type of shared operations that can be performed on the file. The defaults for these options depend on the OPEN attributes, as follows:

OPEN Attribute	Default Sharing
INPUT	SHARED_READ
OUTPUT	NO_SHARE
UPDATE	NO_SHARE

You override these defaults by specifying options for ENVIRONMENT. For example, if SHARED_READ is specified on an OPEN statement for a file opened for UPDATE, the process that opened the file is the only legal writer of the file. Other processes can 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 can write the file. Both the SHARED_READ and SHARED_WRITE options can be specified for a file.

Table 7–3 summarizes the effects of opening a file with file-sharing options.

Table 7–3 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 UNDEFINEDFILE condition is signaled.[2]

[1]You must have write access privileges to open the file with the NO_SHARE option.

[2]ONCODE returns the value for RMS$_FLK.

(continued on next page)

Table 7–3 (Cont.) 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(SHARED_READ) INPUT	ENV(NO_SHARE)	None. The UNDEFINEDFILE 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 UNDEFINEDFILE condition is signaled.[2]
	ENV(SHARED_READ)	None. The UNDEFINEDFILE 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 UNDEFINEDFILE condition is signaled.[2]
	ENV(SHARED_READ)	None. The UNDEFINEDFILE condition is signaled.[2]
	ENV(SHARED_ WRITE)	The file can be accessed for input, output, or update.

[2]ONCODE returns the value for RMS$_FLK.

7.3.2.2 File Locking

If a file is first opened by a process in a manner that restricts sharing by other processes, RMS locks the file to prohibit access by other processes. If a PL/I procedure attempts to open a file already opened by another process for a type of access not allowed, the UNDEFINEDFILE condition is signaled. In an ON-unit that is executed for this condition, the ONCODE built-in function returns the value associated with the RMS status code RMS$_FLK (meaning that the file is locked).

In an application where files must be shared and the synchronization of sharing is important, a procedure can test whether a file is currently being accessed by another process and act accordingly. The following example illustrates an ON-unit that tests whether a file is locked:

```
ON UNDEFINEDFILE(STATE_FILE) BEGIN;
  %INCLUDE $RMSDEF;
    IF ONCODE() = RMS$_FLK
    THEN
       CALL WAITSYNC;
    ELSE
       CALL RESIGNAL();
END;
```

This ON-unit declares the symbolic name RMS$_FLK from PLI$STARLET.TLB and uses an IF statement to verify whether the error occurred because the file is currently locked. If so, the ON-unit calls the procedure WAITSYNC, which presumably synchronizes the procedure's execution. Otherwise, it calls the RESIGNAL built-in subroutine, to request that the default PL/I ON-unit handle the UNDEFINEDFILE condition.

7.3.2.3 Record Locking

When more than one process is accessing a file at the same time, it is necessary to ensure that no process can access a record while another process is writing, rewriting, or deleting the record. To protect access to records in a shared file, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP use the automatic record-locking capability of RMS. Manual record locking is also available, through the record-locking options of the READ statement (LOCK_ON_READ, LOCK_ON_WRITE, MANUAL_UNLOCKING, NOLOCK, NONEXISTENT_RECORD, READ_REGARDLESS, TIMEOUT_PERIOD, and WAIT_FOR_RECORD). These manual record-locking options are valid for all file organizations.

RMS-controlled record locking in PL/I occurs, for example, when one PL/I procedure executes a READ statement (without any record-locking options specified) for a record in a file opened with the UPDATE attribute. During the execution of the READ statement, RMS automatically keeps the record locked, and no other processes can access the record until it is freed.

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

- A READ statement is issued for the record (without any record-locking options specified).
- The file containing the record was opened with the OUTPUT or UPDATE attribute.

A record can also be locked by any of the following options on the READ statement:

LOCK_ON_READ
LOCK_ON_WRITE
MANUAL_UNLOCKING
NONEXISTENT_RECORD
READ_REGARDLESS (on a record that is not already locked)

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 FREE built-in subroutine is called to free all locked records in the file.
- The RELEASE built-in subroutine is called to unlock the record.
- The file is closed.

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

If a procedure in another process attempts to access a record that is locked, the ERROR condition is signaled. In an ON-unit that is executed once this condition exists, 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;
    %INCLUDE $RMSDEF;
    IF ONCODE() = RMS$_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 an attempt to access a locked record. If so, the ON-unit 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.

7.3.2.4 Examples of File Sharing

The following examples illustrate some of the principles of file sharing in VAX PL/I. The procedure UPDATE_FILE obtains, modifies, and rewrites a record in a keyed file. It opens the file with the UPDATE attribute and with the ENVIRONMENT option SHARED_READ. It contains these statements:

```
UPDATE_FILE: PROCEDURE OPTIONS(MAIN);

OPEN FILE(PARTS) RECORD UPDATE KEYED ENV(
              SHARED_READ);
   .
   .
   .
READ FILE(PARTS) INTO(PARTLIST) KEY(INPUT_NUM);
   .
   .
   .
REWRITE FILE(PARTS) FROM(PARTLIST);
```

The procedure PRINT_DATA reads the records in a keyed file sequentially, and displays certain fields in each record. It contains these statements:

```
PRINT_DATA: PROCEDURE OPTIONS(MAIN);
OPEN FILE(PARTS) RECORD INPUT SEQUENTIAL ENV(
                           SHARED_WRITE);
   .
   .
   .
READ FILE(PARTS) INTO(PARTLIST);
DO WHILE (^EOF);
   PUT SKIP LIST(PARTLIST.NAME,QUANTITY.IN_STOCK);
   READ FILE(PARTS) INTO(PARTLIST);
END;
```

The procedure VIEW_DATA reads the records in a keyed file sequentially, and displays certain fields in each record. It does not modify any of the records; it only needs to read them, so it uses the READ statement options READ_REGARDLESS and NOLOCK. It contains these statements:

```
VIEW_DATA: PROCEDURE OPTIONS(MAIN);
OPEN FILE(PARTS) RECORD INPUT SEQUENTIAL ENV(
                            SHARED_WRITE);
   .
   .
   .
READ FILE(PARTS) INTO(PARTLIST) OPTIONS(READ_REGARDLESS,NOLOCK);
DO WHILE (^EOF);
   PUT SKIP LIST(PARTLIST.NAME,QUANTITY.IN_STOCK);
   READ FILE(PARTS) INTO(PARTLIST) OPTIONS(READ_REGARDLESS,NOLOCK);
END;
```

For the purposes of these three examples, assume that the file PARTS is equated to the same OpenVMS file by logical name assignments so that each procedure is attempting access to the same file.

If the process running the program UPDATE_FILE is the first process to open the file, the file is opened for read sharing. When PRINT_DATA or VIEW_DATA opens the file with the SHARED_WRITE option, the file's attribute list indicates that other processes may be writing the file.

If these procedures are executing concurrently, and if, for example, UPDATE_FILE is processing a record in the file PARTS while PRINT_DATA is reading the file, it may happen that PRINT_DATA attempts access to the record being processed by UPDATE_FILE. In this case, the ERROR condition is signaled with the status code RMS$_RLK. But if VIEW_DATA attempts to read a record that is locked by UPDATE_FILE or PRINT_DATA, it can still access the record, because the READ_REGARDLESS option was specified. If VIEW_DATA is reading a record, the other two processes can still access the record, because NOLOCK was also specified as an option to the READ statement.

7.4 ENVIRONMENT Options for I/O Optimization

Many of the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP options for the ENVIRONMENT attribute provide optimization features for I/O operations. Table 7–4 summarizes the options that control disk file allocation. These options let you specify the space requirements of a file when you create it. Table 7–5 summarizes the options for run-time optimization of I/O processing.

Table 7–4 ENVIRONMENT Options for Optimized Disk File Creation

Option	Meaning
BUCKET_SIZE	Specifies the number of disk blocks per bucket, where a bucket is a unit of data storage and transfer.
CONTIGUOUS	Requests that a file's extents be contiguous.
CONTIGUOUS_BEST_TRY	Requests contiguous extents, if possible.
EXTENSION_SIZE	Defines a default extension quantity for the file; to be used whenever the file is enlarged.
FILE_SIZE	Specifies the initial number of disk blocks to be allocated for the file.

Table 7–5 ENVIRONMENT Options for Run-Time Optimization of Input/Output

Option	Meaning
DEFERRED_WRITE	Requests that buffers not be written out until they are full.
MULTIBLOCK_COUNT	Requests multiple blocks for sequential I/O.
MULTIBUFFER_COUNT	Requests multiple buffers for I/O operations.
READ_AHEAD	Requests input and computation overlap for sequential input.
RETRIEVAL_POINTERS	Overrides the default number of file pointers used for file access.
WRITE_BEHIND	Requests output and computation overlap for sequential output.

8

Input/Output Statement Options

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

An I/O statement option remains in effect only for the duration of the statement on which it is specified. The only exception to this rule is the INDEX_NUMBER option.

8.1 Option Format

I/O statement options are specified in a statement by the OPTIONS keyword and an options list. You enclose the options list in parentheses, and separate individual options by commas, as follows:

OPTIONS (option, . . . ;

Following is an example of the I/O statement GET with three options, PROMPT, NO_ECHO, and PURGE_TYPE_AHEAD:

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

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

option(Boolean-expression)

where Boolean-expression is a bit string of length 1,'1'B for true or '0'B for false. If no Boolean expression is specified and the option is present in the option list, the default value of true is supplied.

8.2 Summary of Input/Output Statement Options

Table 8–1 lists the I/O options, briefly describes their uses, indicates which statements they are valid for, and gives their data types.

Table 8–1 Summary of Input/Output Statement Options

Option	Usage	Valid Statements	Data Type
CANCEL_CONTROL_O	Disables effect of Ctrl/o prior to terminal output.	PUT	BIT(1)

(continued on next page)

Table 8–1 (Cont.) Summary of Input/Output Statement Options

Option	Usage	Valid Statements	Data Type
FAST_DELETE	Deletes a record without updating alternate indexes.	DELETE	BIT(1)
FIXED_CONTROL_FROM (variable)	Modifies the fixed control area of a record.	REWRITE WRITE	Data type of fixed control area
FIXED_CONTROL_TO (variable)	Returns the contents of the fixed control area	READ	Data type of fixed control area
INDEX_NUMBER (expression)	Specifies the index to which an I/O operation applies.	DELETE READ REWRITE	FIXED BINARY(31)
LOCK_ON_READ	Locks a record for reading and allows other readers but no writers.	READ	BIT(1)
LOCK_ON_WRITE	Locks a record for writing and allows other readers but no writers.	READ	BIT(1)
MANUAL_UNLOCKING	Specifies that the user, not RMS, is to control record locking and unlocking.	READ	BIT(1)
MATCH_GREATER	Matches any key with a value greater than the value of the KEY option.	DELETE READ REWRITE	BIT(1)
MATCH_GREATER_EQUAL	Matches any key with a value greater than or equal to the value of the KEY option.	DELETE READ REWRITE	BIT(1)
MATCH_NEXT	Matches any key with a value greater than the value of the KEY option.	DELETE READ REWRITE	BIT(1)
MATCH_NEXT_EQUAL	Matches any key with a value greater than or equal to the value of the KEY option.	DELETE READ REWRITE	BIT(1)
NO_ECHO	Suppresses display of input data on a terminal.	GET	BIT(1)
NO_FILTER	Suppresses recognition of Ctrl/u, Ctrl/r, and the DEL key on input operations.	GET	BIT(1)
NOLOCK	Disables record locking for the current operation.	READ	BIT(1)
NONEXISTENT_RECORD	Locks a nonexistent record.	READ	BIT(1)
PROMPT (expression)	Writes a prompting message prior to an input operation.	GET	CHAR(*)
PURGE_TYPE_AHEAD	Clears a terminal's type-ahead buffer before reading input data.	GET	BIT(1)

(continued on next page)

Table 8–1 (Cont.) Summary of Input/Output Statement Options

Option	Usage	Valid Statements	Data Type
READ_REGARDLESS	Enables a record to be read regardless of any lock.	READ	BIT(1)
RECORD_ID (variable)	Accesses a record based on its internal identification.	DELETE READ REWRITE	(2) FIXED BINARY(31)
RECORD_ID_TO (variable)	Returns the value of a record's internal identification.	READ REWRITE WRITE	(2) FIXED BINARY(31)
TIMEOUT_PERIOD (expression)	Avoids a potential deadlock by indicating the number of seconds to wait before returning an error; used only with WAIT_FOR_ RECORD.	READ	FIXED BINARY(31)
WAIT_FOR_RECORD	If a record is locked, causes the process to wait until it is available.	READ	BIT(1)

8.2.1 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 before data is output. This ensures that the beginning of the output list is displayed.

Rules

- The CANCEL_CONTROL_O option is valid only on a PUT statement.
- 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 displayed 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 has been suspended by Ctrl/o prior to execution of the PUT statement, the CANCEL_CONTROL_O option on the PUT statement cancels the effect of the Ctrl/o and outputs the data list.

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

Rules

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

Usage

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

8.2.3 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 as follows:

FIXED_CONTROL_FROM (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

- The FIXED_CONTROL_FROM option is valid on the WRITE and REWRITE statements.
- The file must have variable-length records with a fixed-length control area and must be opened with the OUTPUT or UPDATE attribute. If the file is opened with the OUTPUT attribute, the ENVIRONMENT option FIXED_CONTROL_SIZE must also be specified.
- The length of the variable must match the length of the fixed control area, as specified in the FIXED_CONTROL_SIZE option of ENVIRONMENT. If the variable is not of the correct length, the ERROR condition is signaled.

Usage

The following example illustrates writing a file with sequence numbers in a fixed control area:

```
DECLARE (OUTFILE,INFILE) FILE,
        LINE_NUM FIXED BINARY(15), /* sequence numbers */
        COPY_REC CHARACTER(132) VARYING,
        EOF BIT(1) STATIC INIT('0'B);
OPEN FILE(INFILE) INPUT SEQUENTIAL;
ON ENDFILE(INFILE) EOF = '1'B;
OPEN FILE (OUTFILE) OUTPUT RECORD SEQUENTIAL
         ENVIRONMENT (FIXED_CONTROL_SIZE(2));
READ FILE(INFILE) INTO(COPY_REC);
/* Increment sequence number; copy record to output file */
DO LINE_NUM = 100 BY 100 WHILE (^EOF);
       WRITE FILE(OUTFILE) FROM (COPY_REC)
            OPTIONS(FIXED_CONTROL_FROM (LINE_NUM));
       READ FILE(INFILE) INTO(COPY_REC);
       END;
CLOSE FILE (INFILE);
CLOSE FILE(OUTFILE) ENVIRONMENT (SPOOL);
```

In this example, the OpenVMS file associated with the PL/I file OUTFILE will have a 2-byte fixed control area. Line numbers are assigned in increments of 100. Note that a file in this format, that is, a file that has CARRIAGE_RETURN_FORMAT carriage control (the default) and a 2-byte fixed control area, is handled in a special way by the OpenVMS system. When this file is printed with the DCL command PRINT or queued by the SPOOL option (as in this example), the contents of the fixed control area are printed to the left of each record on the output listing.

8.2.4 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 as follows:

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

- The FIXED_CONTROL_TO option is valid only on a READ statement.
- 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.
- 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.

8.2.5 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 as follows:

INDEX_NUMBER (integer-expression)

integer-expression
Specifies the index to be used. The value of the integer expression must be the number of an index for records in an indexed sequential file. The primary index is 0, the secondary index is 1, and so on.

Rules

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

Usage
The INDEX_NUMBER option on an I/O statement overrides the current index number, which can be set explicitly by the INDEX_NUMBER option of ENVIRONMENT or implicitly by a WRITE statement that specifies the KEY option or the RECORD_ID option.

When the INDEX_NUMBER option is used, the specified index becomes the current index for the file and is used in this and in all subsequent I/O operations until the INDEX_NUMBER option is again specified. For example:

```
GET LIST(BIRD) OPTIONS (PROMPT('Enter bird'));

READ FILE(STATEFILE) INTO(STATE) KEY(BIRD)
           OPTIONS (INDEX_NUMBER(2));
```

In this example, the READ statement accesses the record in the file STATEFILE using the index numbered 2.

8.2.6 LOCK_ON_READ Option

The LOCK_ON_READ option specifies a lock for reading that allows other readers but no writers. If you specify this option, then a record stream with a shared file that is open for reading only, is permitted to lock a record from modification by other programs or streams. Other streams are permitted to read the record but not to lock it.

Rules

- The LOCK_ON_READ option is valid only on a READ statement.
- This option conflicts with the NOLOCK option.
- This option remains in effect only for the current statement; then it is reset to false.

8.2.7 LOCK_ON_WRITE Option

The LOCK_ON_WRITE option specifies that a record will be locked for possible modifications. However, readers will be able to access the record. Streams that are locking records for modification can therefore allow nonlocking streams to read locked records.

Rules

- The LOCK_ON_WRITE option is valid only on a READ statement.
- This option conflicts with the NOLOCK option.
- This option remains in effect only for the current statement; then it is reset to false.

8.2.8 MANUAL_UNLOCKING Option

The MANUAL_UNLOCKING option specifies that a record will be locked until it is explicitly unlocked by the process, thus giving you (instead of RMS) control over locking and unlocking.

Rules

- The MANUAL_UNLOCKING option is valid only on a READ statement.
- This option conflicts with the NOLOCK option.
- This option remains in effect only for the current statement; then it is reset to false.

8.2.9 MATCH_NEXT Option

The MATCH_NEXT option specifies that the record of interest is the first record whose key is greater than the key specified in the KEY option. MATCH_NEXT overrides the default rule for key matching, which is to look for an exact key match.

MATCH_GREATER is an obsolete synonym for MATCH_NEXT.

Rules

- The MATCH_NEXT option is valid on the READ, REWRITE, and DELETE statements.
- The KEY option must also be specified.
- The file must be an indexed sequential file or a relative file.

- The MATCH_NEXT option conflicts with the MATCH_NEXT_EQUAL option.
- The MATCH_NEXT option remains in effect only for the current statement; then it is reset to false.

Usage

In the following example, STATE_FILE's third alternate key (that is, index number 3) 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 LIST(SIZE) OPTIONS(PROMPT(
       'Population value: '));
READ FILE(STATE_FILE) INTO(STATE) KEY(SIZE)
       OPTIONS(MATCH_NEXT,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 can 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 3 whose key value is greater than 8000000.

8.2.10 MATCH_NEXT_EQUAL Option

The MATCH_NEXT_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.

MATCH_GREATER_EQUAL is an obsolete synonym for MATCH_NEXT_EQUAL.

Rules

- The MATCH_NEXT_EQUAL option is valid on the READ, REWRITE, and DELETE statements.
- The KEY option must also be specified.
- The file must be an indexed sequential file or a relative file.
- The MATCH_NEXT_EQUAL option conflicts with the MATCH_NEXT option.
- The MATCH_NEXT_EQUAL option remains in effect only for the current statement; then it is reset to false.

8.2.11 NO_ECHO Option

The NO_ECHO option specifies, when the input device is a terminal, that the data entered at the terminal will not be displayed as it is entered.

Rules

- The NO_ECHO option is valid only on a GET statement.
- This option is ignored if the input device is not a terminal.
- This option remains in effect only for the current statement; then it is reset to false.

Usage

This option is useful when data entered at a terminal is to be protected from being seen by users other than the one who entered the data. For example, if a password is to be entered, the NO_ECHO option protects the password, as follows:

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

Data entered in response to this GET statement is not displayed on the terminal.

8.2.12 NO_FILTER Option

The NO_FILTER option specifies, when the input device is a terminal, that the recognition of Ctrl/u, Ctrl/r, and the DEL key is to be suppressed. These characters are interpreted as terminators.

Rules

- The NO_FILTER option is valid only on a GET statement.
- This option is ignored if the input device is not a terminal.
- This option remains in effect only for the current statement; then it is reset to false.

Usage

When NO_FILTER is in effect, the terminal keys that normally permit a user to edit data as it is entered do not perform their normal functions. For example:

123 DEL

If this data is entered in response to a GET statement that specifies the NO_FILTER option, the DEL key does not delete the last character typed (3); instead, it acts as the terminator of the input, just as the RETURN key usually does, and the value 123 is assigned to the input variable.

8.2.13 NOLOCK Option

The NOLOCK option specifies that a record accessed with a READ statement is not to be locked during the current operation.

Rules

- The NOLOCK option is valid only on a READ statement.
- This option should not be used if the record is to be updated or deleted, because an attempt to perform either one of these operations on an unlocked record will fail.
- The NOLOCK option conflicts with the LOCK_ON_READ, LOCK_ON_WRITE, and MANUAL_UNLOCKING options.
- The NOLOCK option remains in effect only for the current statement; then it is reset to false.

Usage

The NOLOCK option can improve the speed of reading if it is used on a file that is opened with the ENVIRONMENT option SHARED_READ or with both SHARED_READ and SHARED_WRITE.

8.2.14 NONEXISTENT_RECORD Option

The NONEXISTENT_RECORD option locks a randomly accessed record that does not exist in the file at the time of access. It prevents other streams from putting a new record into that cell until the stream that locked it either puts a record there itself or releases the record lock.

Rules

- The NONEXISTENT_RECORD option is valid only on a READ statement.
- This option applies only to relative files.
- This option remains in effect only for the current statement; then it is reset to false.

8.2.15 PROMPT Option

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

PROMPT (string-expression)

string-expression

Specifies a 1- to 254-character string expression.

Rules

- The PROMPT option is valid only on a GET statement.
- 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 Return
```

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

On a terminal, using the PROMPT option provides the following benefits:

- 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.
- 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 you specify the SKIP option in conjunction with the PROMPT option, the SKIP operation is performed before the prompting message is displayed.

8.2.16 PURGE_TYPE_AHEAD Option

When the input device is a terminal, the PURGE_TYPE_AHEAD option specifies that all data in the terminal's type-ahead buffer be deleted before the input operation is performed.

Rules

- The PURGE_TYPE_AHEAD option is valid only on a GET statement.
- This option is ignored if the input device is not a terminal.

Usage

When a command or program is being executed, the terminal keyboard accepts input data and stores it in a buffer called the type-ahead buffer. When the command or program is completed, the command interpreter reads its next command from the type-ahead buffer. When a GET statement is executed with this option in effect, any data in the type-ahead buffer is deleted, ensuring that the GET statement will not read any extraneous data.

8.2.17 READ_REGARDLESS Option

The READ_REGARDLESS option allows a record to be read regardless of whether it is locked.

Rules

- The READ_REGARDLESS option is valid only on a READ statement.
- The READ_REGARDLESS option conflicts with the WAIT_FOR_RECORD option.
- If the record is not already locked, the READ_REGARDLESS option locks it.
- The READ_REGARDLESS option remains in effect only for the current statement; then it is reset to false.

8.2.18 RECORD_ID Option

The RECORD_ID option indicates that the record of interest is specified by its record identification. The format of this option is as follows:

```
RECORD_ID (variable-reference)
```

variable-reference

Specifies the name of a 2-element array variable containing the record identification.

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

Rules

- The RECORD_ID option is valid on a READ, REWRITE, or DELETE statement.
- RECORD_ID conflicts with the KEY option on the READ, REWRITE, or DELETE statement.
- The file on which the operation is being performed must have been opened with the ENVIRONMENT option RECORD_ID_ACCESS.
- If the file is an indexed sequential file, the RECORD_ID option resets the value of the current index number to 0.

Usage

The following example illustrates a record whose record identification is saved for later file access.

```
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(SAVE_RECORD_ID));
```

During the first opening 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.

8.2.19 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 as follows:

RECORD_ID_TO (variable-reference)

variable-reference

Is a reference to a 2-element array variable that will receive the value of the record's identification.

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

Rules

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

8.2.20 TIMEOUT_PERIOD Option

The TIMEOUT_PERIOD option, used only with the WAIT_FOR_RECORD option, causes the waiting condition to continue only for the specified timeout period, in seconds. If the timeout period expires before the lock is granted, an error is signaled.

Rules

- The TIMEOUT_PERIOD option is valid only on a READ statement.
- If the TIMEOUT_PERIOD option is specified without the WAIT_FOR_RECORD option, it is ignored.
- The timeout period must be between 0 and 255 seconds.
- The TIMEOUT_PERIOD option remains in effect while the WAIT_FOR_RECORD option remains in effect, that is, for the current statement only.

Usage

The TIMEOUT_PERIOD option prevents the WAIT_FOR_RECORD option from potentially causing an indefinite deadlock in the process.

In the following example, a 10-second waiting period is specified for a locked record. If the record is still locked after that period expires, an error is signaled.

```
READ FILE(DATAFILE) INTO (BUFFER)
   OPTIONS(WAIT_FOR_RECORD,TIMEOUT_PERIOD(10));
```

8.2.21 WAIT_FOR_RECORD Option

The WAIT_FOR_RECORD option specifies that if a record is already locked, the process will wait until the record is available.

Rules

- The WAIT_FOR_RECORD option is valid only on a READ statement.
- The WAIT_FOR_RECORD option conflicts with the READ_REGARDLESS option.
- The WAIT_FOR_RECORD option remains in effect only for the current statement; then it is reset to false.

Usage

The WAIT_FOR_RECORD option can be used with the TIMEOUT_PERIOD option to avoid an indefinite wait.

9
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 Record Management Services (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 9–1 and are described individually in the following sections.

Table 9–1 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.
FREE	Unlocks all the locked records in a file.
NEXT_VOLUME	Begins processing the next volume in a multivolume tape set.
RELEASE	Unlocks a specified record in a file.
REWIND	Positions a file at its beginning or at a specific record.
SPACEBLOCK	Positions a file forward or backward a specified number of blocks.

9.1 DISPLAY Built-In Subroutine

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

```
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.

The format of the data returned by DISPLAY is defined in the data structure PLI_FILE_DISPLAY. This structure is declared in the text module PLI_FILE_DISPLAY in the default INCLUDE library PLI$STARLET (the PL/I compiler searches this library by default when it compiles a PL/I program). Each member of PLI_FILE_DISPLAY contains, on return from a call to DISPLAY, a value associated with the file for which information is requested. To refer to a

value, you refer to the corresponding member name in the structure. Tables 9–2 through 9–4 summarize the members of the structure as follows:

- Members containing information about the settings of ENVIRONMENT options
- Members containing information on file attributes
- Members containing information on device attributes

You declare the structure PLI_FILE_DISPLAY with the BASED attribute; thus, to use this variable you must also declare a pointer variable to reference the structure and use an ALLOCATE statement to allocate storage for it before calling DISPLAY. For example:

```
%INCLUDE PLI_FILE_DISPLAY;
DECLARE STATE_FILE FILE RECORD KEYED,
        FILEPTR POINTER;
        OPEN FILE(STATE_FILE);
        ALLOCATE PLI_FILE_DISPLAY SET (FILEPTR);
        CALL DISPLAY (STATE_FILE,FILEPTR->PLI_FILE_DISPLAY);
```

Following this call to DISPLAY, you can reference any of the members of FILEPTR–>PLI_FILE_DISPLAY to determine information about the file STATE_FILE. The following statements use the EXPANDED_TITLE field to display the expanded file specification of STATE_FILE and the INDEXED and NUMBER_OF_KEYS fields to display the number of keys in the file:

```
PUT SKIP EDIT('File',FILEPTR->EXPANDED_TITLE,
                              'opened for input')
             (A,X,A,X,A);
IF FILEPTR->INDEXED THEN PUT SKIP EDIT
      ('It is indexed with',FILEPTR->NUMBER_OF_KEYS,'keys')
      (A,X,A,X,A);
```

If you do not use the structure PLI_FILE_DISPLAY, as shown in this example, you must provide a structure that has the same declaration as PLI_FILE_DISPLAY. To obtain a copy of PLI_FILE_DISPLAY, use the LIBRARY command. For example:

```
$ LIBRARY/TEXT/EXTRACT=PLI_FILE_DISPLAY/OUTPUT=FILESTRUC.PLI -
$_SYS$LIBRARY:PLI$STARLET
```

Here, FILESTRUC.PLI is the name of the output file into which the LIBRARY command will copy PLI_FILE_DISPLAY.

Table 9–2 summarizes the values returned by DISPLAY that correspond to ENVIRONMENT options and the data type of each structure member. For information on ENVIRONMENT options, see Chapter 7.

Table 9–2 ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
APPEND	BIT(1)	APPEND option is enabled or disabled.
BACKUP_DATE	BIT(64) ALIGNED	Backup date of file (disk files only).
BATCH	BIT(1)	BATCH option is enabled or disabled.

(continued on next page)

Table 9–2 (Cont.) ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
BLOCK_BOUNDARY_FORMAT	BIT(1)	Records cannot cross block boundaries.
BLOCK_IO	BIT(1)	File is opened for block I/O.
BLOCK_SIZE	FIXED BIN	Block size of file (magnetic tape files only).
BUCKET_SIZE	FIXED BIN	Bucket size of file (disk files only).
CARRIAGE_RETURN_FORMAT	BIT(1)	Records have carriage return carriage control.
CONTIGUOUS	BIT(1)	CONTIGUOUS option is enabled or disabled.
CONTIGUOUS_BEST_TRY	BIT(1)	CONTIGUOUS_BEST_TRY option is enabled or disabled.
CREATION_DATE	BIT(64) ALIGNED	Creation date of file.
CURRENT_POSITION	BIT(1)	CURRENT_POSITION option is enabled or disabled.
DEFERRED_WRITE	BIT(1)	DEFERRED_WRITE option is enabled or disabled.
DELETE	BIT(1)	DELETE option is enabled or disabled.
EXPIRATION_DATE	BIT(64) ALIGNED	Expiration date of file.
EXTENSION_SIZE	FIXED BIN	Current extension size (disk files only).
FILE_ID	(6) FIXED BIN	File identification (disk files only).
FILE_SIZE	FIXED BIN	File allocation (disk files only).
FIXED_CONTROL_SIZE	FIXED BIN	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 or disabled.
INDEX_NUMBER	FIXED BIN	Current index number.
INDEXED	BIT(1)	File is or is not an indexed sequential file.
INITIAL_FILL	BIT(1)	INITIAL_FILL option is enabled or disabled.
MAXIMUM_RECORD_NUMBER	FIXED BIN	Relative file maximum relative record.
MAXIMUM_RECORD_SIZE	FIXED BIN	Largest record size.
MULTIBLOCK_COUNT	FIXED BIN	Multiblock count (disk files only).
MULTIBUFFER_COUNT	FIXED BIN	Multibuffer count.
NO_SHARE	BIT(1)	NO_SHARE option is enabled or disabled.
OWNER_GROUP	FIXED BIN	Group number of file's owner.
OWNER_MEMBER	FIXED BIN	Member number of file's owner.
OWNER_PROTECTION	CHAR(4) VARYING	Protection for file's owner.
RETRIEVAL_POINTERS	FIXED BIN	Number of mapping pointers.
PRINTER_FORMAT	BIT(1)	Records have printer carriage control.
READ_AHEAD	BIT(1)	READ_AHEAD option is enabled or disabled.
READ_CHECK	BIT(1)	READ_CHECK option is enabled or disabled.

(continued on next page)

Table 9–2 (Cont.) ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
RECORD_ID_ACCESS	BIT(1)	File is opened for access by record identification.
REVISION_DATE	BIT(64) ALIGNED	Revision date of file (disk files only).
REWIND_ON_CLOSE	BIT(1)	REWIND_ON_CLOSE option is enabled or disabled.
REWIND_ON_OPEN	BIT(1)	REWIND_ON_OPEN option is enabled or disabled.
SCALARVARYING	BIT(1)	SCALARVARYING option is enabled or disabled.
SHARED_READ	BIT(1)	SHARED_READ option is enabled or disabled.
SHARED_WRITE	BIT(1)	SHARED_WRITE option is enabled or disabled.
SPOOL	BIT(1)	SPOOL option is enabled or disabled.
SUPERSEDE	BIT(1)	SUPERSEDE option is enabled or disabled.
SYSTEM_PROTECTION	CHAR(4) VARYING	Protection for system users.
TEMPORARY	BIT(1)	TEMPORARY option is enabled or disabled.
TRUNCATE	BIT(1)	TRUNCATE option is enabled or disabled.
WORLD_PROTECTION	CHAR(4) VARYING	Protection for world users.
WRITE_BEHIND	BIT(1)	WRITE_BEHIND option is enabled or disabled.
WRITE_CHECK	BIT(1)	WRITE_CHECK option is enabled or disabled.

Table 9–3 summarizes the file attribute information returned by DISPLAY. All names in the table are level-2 members of the structure PLI_FILE_DISPLAY.

Table 9–3 File Attribute Information Returned by DISPLAY

Member Name	Type of Value Returned	Data Type of Meaning
COLUMN_NUMBER	FIXED BIN	Current column (stream output files only).
DIRECT	BIT(1)	File has or 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 or does not have FTN (ASA) carriage control.
INPUT	BIT(1)	File has or does not have INPUT attribute.
KEYED	BIT(1)	File has or does not have KEYED attribute.
LINE_NUMBER	FIXED BIN	Current line number (stream output files only).
LINESIZE	FIXED BIN	File's line size (stream output files only).

(continued on next page)

Table 9–3 (Cont.) File Attribute Information Returned by DISPLAY

Member Name	Type of Value Returned	Data Type of Meaning
NUMBER_OF_KEYS	FIXED BIN	Number of keys (indexed sequential files only).
OUTPUT	BIT(1)	File has or does not have OUTPUT attribute.
PAGE_NUMBER	FIXED BIN	Current page number (PRINT files only).
PAGESIZE	FIXED BIN	Page size (PRINT files only).
PRINT	BIT(1)	File has or does not have PRINT attribute.
RECORD	BIT(1)	File has or does not have RECORD attribute.
SEQUENTIAL	BIT(1)	File has or does not have SEQUENTIAL attribute.
STREAM	BIT(1)	File has or does not have STREAM attribute.
UPDATE	BIT(1)	File has or does not have UPDATE attribute.

Table 9–4 lists the names of the structure members that contain information about the device to which a file is written or from which the file is to be read. All of the names in Table 9–4 are level-3 members of the structure PLI_FILE_DISPLAY; they each appear within the following minor structures, which have identical declarations:

- DEVICE
- SPOOLING_DEVICE

If the field PLI_FILE_DISPLAY.DEVICE.SPL is true, then the members of the minor structure DEVICE contain information about the device that is spooled. Members of minor structure PLI_FILE_DISPLAY.SPOOLING_DEVICE contain information about the intermediate, or spooling, device.

All fields within these structures are BIT(1) values.

Table 9–4 Device Information Returned by DISPLAY

Member Name	Meaning
ALL	Device is or is not allocated.
AVL	Device is or is not online and available.
CCL	Device has or does not have carriage control.
DIR	Device is or is not directory structured.
DMT	Device is or is not marked for dismounting.
ELG	Device is or is not enabled for error logging.
FOD	Device is or is not file-oriented.
FOR	Device is or is not a foreign device.
GEN	Device is or is not a generic device.
IDV	Device is or is not capable of input.
MBX	Device is or is not a mailbox.
MNT	Device is or is not mounted.
NET	Device is or is not a network device.

(continued on next page)

Table 9–4 (Cont.) Device Information Returned by DISPLAY

Member Name	Meaning
ODV	Device is or is not capable of output.
RCK	Device performs read checking.
REC	Device is or is not a record-oriented device (terminal or line printer, for example).
RND	Device is or is not random access in nature.
RTM	Device is or is not a real-time device.
SDI	Device has a master directory only.
SHR	Device is or is not shareable.
SPL	Device is or is not spooled.
SQD	Device is or is not sequential block-oriented (magnetic tape).
SWL	Device is or is not currently software write-locked.
TRM	Device is or is not a terminal.
WCK	Device performs write checking.

9.2 EXTEND Built-In Subroutine

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

```
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 0 to 4,294,967,295, specifying the number of 512-byte disk blocks to be added 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 interprets the number as an unsigned integer.

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 making 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.

9.3 FLUSH Built-In Subroutine

The FLUSH built-in subroutine writes all RMS buffers that have been modified and preserves all of the file attributes of the file. This subroutine provides the ability to checkpoint a file during its processing and ensure its integrity. Its calling sequence is as follows:

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

file-reference
Specifies the name of the file variable or file constant associated with the file whose buffers are to be flushed. If the file is not currently opened, the FLUSH subroutine performs no operation.

Use the FLUSH subroutine to explicitly request RMS to write all internal file buffers back to the file. This subroutine is called implicitly by the REWIND and NEXT_VOLUME built-in subroutines.

9.4 FREE Built-In Subroutine

The FREE built-in subroutine unlocks all the locked records in a specified file. Its calling sequence is as follows:

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

file-reference
Specifies the name of the file variable or file constant associated with the file whose records are to be unlocked.

9.5 NEXT_VOLUME Built-In Subroutine

The NEXT_VOLUME built-in subroutine performs the positioning and labeling functions necessary when the next volume is required during I/O to a magnetic tape file that spans more than one physical tape volume. Its calling sequence is as follows:

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

file-reference
Specifies the name of the file constant or file variable associated with the tape volume set that is being processed. If the file is not currently open, the NEXT_ VOLUME subroutine implicitly opens the file with the attributes specified in the declaration of the file.

When a multivolume tape file is being read or written, volume switching is normally transparent to the PL/I program. RMS and the magnetic tape Ancillary Control Program (ACP) perform all the steps necessary to ensure that the next required volume is physically mounted, initialized, and verified.

However, when a program must advance to the next volume before reaching the end of the current volume on input, or before the end of the tape is reached on output, it can call the NEXT_VOLUME built-in subroutine. This subroutine performs all the necessary volume checking when a multivolume tape file is being read. When a file is being written, the subroutine writes the appropriate information on the output tapes.

For more detailed explanations of volume switching, see the *OpenVMS Record Management Services Reference Manual* and the *OpenVMS DCL Dictionary*.

9.6 RELEASE Built-In Subroutine

The RELEASE built-in subroutine unlocks a specified record in a file. Its calling sequence is as follows:

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

file-reference
Specifies the name of the file variable or file constant associated with the file on which the operation is to be performed. This file must have been opened with the RECORD_ID_ACCESS option of the ENVIRONMENT attribute.

variable-reference
Specifies the record identification of the record to be unlocked. It must be declared (2) FIXED BINARY, and it must be connected.

9.7 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 as follows:

```
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.

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:

- A sequential file is positioned at its first record.
- A relative record is positioned at the first occupied cell.
- An indexed sequential file is positioned at the lowest key value in the current index.
- A tape file on a single volume is rewound; a tape file on a multivolume tape set is rewound 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 any more data can be read. However, an ENDFILE ON-unit can be written as follows:

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

This ON-unit calls the REWIND built-in subroutine each time an end-of-file condition 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.

9.8 SPACEBLOCK Built-In Subroutine

The SPACEBLOCK built-in subroutine positions a file forward or backward a specified number of blocks. This subroutine can be used to process unlabeled magnetic tapes, as well as sequential disk files that are being processed with block I/O. Its calling sequence is as follows:

```
CALL SPACEBLOCK (file-reference,integer-expression);
```

file-reference
Specifies the name of the file constant or file variable that is to be spaced. If the file is open, it must have been opened with the BLOCK_IO option. If the file is not open, the SPACEBLOCK subroutine opens the file with the BLOCK_IO option.

integer-expression
Is a fixed binary expression specifying the number of blocks to be spaced forward or backward. If the expression is negative, the file is spaced backward the specified number of blocks. If the expression is positive, the tape is spaced forward the specified number of blocks.

10
Error Handling

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, errors are signaled through ON conditions and handled by groups of statements called ON-units. An ON condition is any one of several named conditions that interrupt program execution. When an ON condition occurs, or is signaled, the corresponding ON-unit is executed.

This chapter discusses the following topics:

- The RESIGNAL built-in subroutine
- The actions that ON-units can take
- The relationship of OpenVMS condition handling to PL/I for OpenVMS VAX and PL/I for OpenVMS AXP condition handling
- The search for ON-units when a condition is signaled, and the default handling performed when no ON-unit exists
- The scope of ON-units
- Some examples of ON-units
- The condition-handling built-in functions (ONARGSLIST, ONCODE, ONFILE, and ONKEY)

Refer to the *PL/I for OpenVMS Systems Reference Manual* for descriptions of the ON, REVERT, and SIGNAL statements and ON conditions.

10.1 RESIGNAL Built-In Subroutine

The RESIGNAL built-in subroutine is used in an ON-unit to pass a signaled condition, so that the run-time system will attempt to locate another ON-unit to handle the condition.

RESIGNAL works by setting up the internal mechanism for passing the signal. It does not by itself cause an exit from the ON-unit that calls it. Instead, it returns to the next statement in the ON-unit. Resignaling does not occur until execution of the ON-unit is completed.

The format of a statement calling the RESIGNAL built-in subroutine is as follows:

```
CALL RESIGNAL();
```

When an ON-unit has determined that it cannot or should not respond to a condition, RESIGNAL permits the ON-unit to pass the signal along.

This subroutine is not part of the standard PL/I language. It is provided specifically for use in the OpenVMS operating system environment.

10.2 ON-Unit Actions

During its execution, an ON-unit can take any of the following courses of action:

- 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 in the following subsections.

10.2.1 Handling the Condition

A condition is assumed to be handled in PL/I when the ON-unit established for the condition completes execution 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. Normal completion of any ON-unit (except ERROR signaled as the default action) results in return of control either to the statement that caused the condition or to the statement immediately following the statement that caused the condition. However, the effects of normal return from ERROR, FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVIDE are generally unpredictable. Exceptions are cases of ERROR that are specifically documented to allow normal return, and ON-units that execute as a result of a SIGNAL statement. In the case of UNDERFLOW, return from the default PL/I condition handler continues execution unpredictably, with an undefined value as the result of the operation that caused the condition (the value is set to zero only if the UNDERFLOW option is not enabled).

10.2.2 Resignaling the Condition

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, an ON-unit can choose not to handle a condition and can request that, rather than returning control to the point of interruption, PL/I continue to search for another ON-unit to handle the condition. It does this by calling the RESIGNAL built-in subroutine as follows:

```
CALL RESIGNAL();
```

The RESIGNAL built-in subroutine has no arguments.

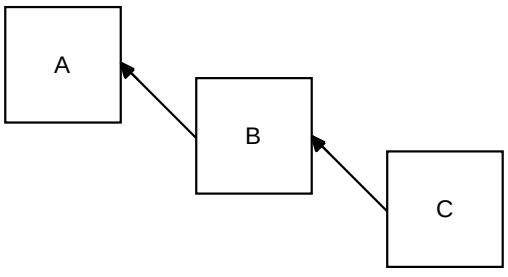
When an ON-unit calls RESIGNAL, PL/I resumes its search of the call stack starting at the call frame beneath the frame in which it located the current ON-unit. Example 10–1 and Figure 10–1 illustrate the effect of the RESIGNAL built-in subroutine. The callout numbers in the example indicate the order of execution.

Procedure B establishes an ANYCONDITION ON-unit that handles specific VAXCONDITION values. When the ON-unit is executed as a result of a signal in either procedure B or C, it tests the current value of ONCODE to see whether it is a value of interest. If not, procedure B calls RESIGNAL.

Example 10–1 Resignaling the Condition

```
A: PROCEDURE OPTIONS (MAIN);1
ON FIXEDOVERFLOW BEGIN;

6

END;
   .
   .
   .
CALL B;

B: PROCEDURE (X,Y);2
ON ANYCONDDITION BEGIN;
IF (ONCODE()=VAXCONDITION(SIGNAL_FOUND))!5
   (ONCODE()=VAXCONDITION(SIGNAL_DONE)) THEN
   BEGIN;

   END;/"begin block for IF statement"/
   ELSE CALL RESIGNAL();
END;/"ON-unit"/
CALL C;

C: PROCEDURE;3

    4

   RETURN;
   END;
```

Figure 10–1 Resignaling a Condition



NU−2481A−RA

When the FIXEDOVERFLOW condition is signaled in procedure C, B calls RESIGNAL and PL/I continues its search of the call stack. It locates the ON-unit for handling the FIXEDOVERFLOW condition in procedure A and executes it.

Note that the default condition handling performed by PL/I uses the resignaling capability to continue signals that are not handled within the program. PL/I default condition handling is described in Section 10.4.

10.2.3 Unwinding the Call Stack

An ON-unit in a PL/I procedure can execute a nonlocal GOTO statement that transfers control to a previous block. In this case, PL/I releases call frames, beginning with the call frame created for the ON-unit, until it reaches the block containing the label specified in the GOTO statement.

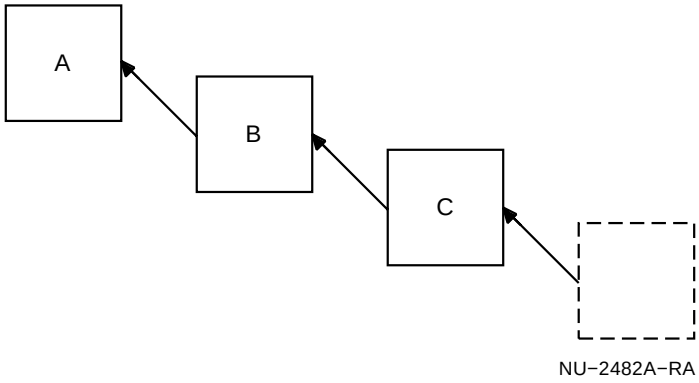
The removal of call frames from the call stack is called an unwind. Example 10–2 and Figure 10–2 illustrate a situation in which an unwind occurs. The callout numbers in the example indicate the order of execution.

Procedure A establishes an ERROR ON-unit represented by the box drawn with broken lines in Figure 10–2. 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.

Example 10–2 Unwinding the Call Stack

```
A: PROCEDURE OPTIONS (MAIN);1
       ON ERROR GOTO PRINT_MSG;

       4

       CALL B;
   PRINT_MSG;

   5

   END;

B: PROCEDURE;2

   CALL C;

   RETURN;
   END;

C: PROCEDURE;3

   RETURN;
   END;
```

Figure 10–2 Unwinding the Call Stack



When an unwind occurs in the OpenVMS 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.

10.2.4 Stopping the Program

An ON-unit can 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;
```

The STOP statement performs the following actions:

- It signals the FINISH condition.
- It calls the SYS$EXIT system service to perform an image exit.

Thus, when a FINISH ON-unit that has been executed as a result of a STOP statement handles the condition, control returns to the STOP statement, which then terminates the image.

If no FINISH or ANYCONDITION ON-unit exists, the program is terminated.

Note that when a FINISH or ANYCONDITION ON-unit executes a STOP statement, the program enters an infinite loop.

10.3 Relationship of OpenVMS Condition Handlers to PL/I ON-Units

In the OpenVMS 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 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 PL/I

for OpenVMS VAX and PL/I for OpenVMS AXP, you can write ON-units to handle any possible OpenVMS-specific condition. Each procedure can establish separate ON-units for each of several possible conditions that the procedure wishes to handle.

You should never use LIB$ESTABLISH to establish a condition handler in a PL/I call frame.

10.4 Search Path for ON-Units

When a condition is signaled in the OpenVMS environment, the OpenVMS condition-handling facility searches the call stack, beginning with the call frame within which the condition was signaled, for a condition handler. If there is no handler, or if no handler handles the condition, a system default handler is executed.

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, the search rules are different. PL/I searches each call frame in the calling sequence for ON-units in a specific sequence. If it reaches the call frame at which the program was entered without locating an ON-unit, it performs default condition handling. The default handling depends on whether or not the call frame at which the procedure was entered specified the MAIN option.

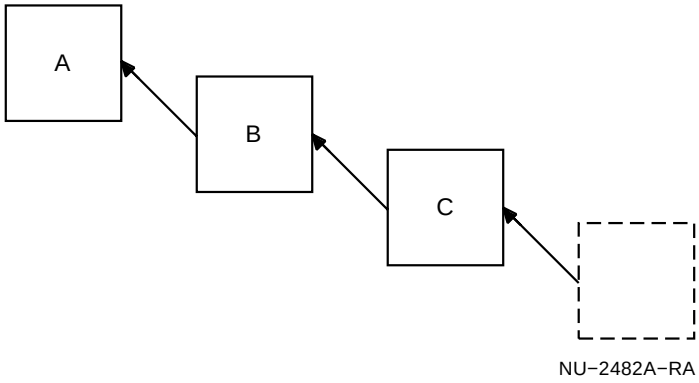
Example 10–3 and Figure 10–3 illustrate the call frames established for the execution of a series of procedures. The callout numbers in the example indicate the order of execution.

The box drawn with broken lines in Figure 10–3 represents the ON-unit established in procedure A for a FIXEDOVERFLOW condition. Procedure A establishes an ON-unit for the FIXEDOVERFLOW condition, and procedure B establishes an ON-unit for an UNDEFINEDFILE condition for the file PRINTFILE. When a FIXEDOVERFLOW condition is signaled in procedure C, PL/I locates the ON-unit established in procedure A and activates the corresponding ON-unit. When PL/I activates an ON-unit, it creates an activation record for the ON-unit and places the ON-unit on the call stack for execution, as if it were a unique block activation.

Example 10–3 Execution of an ON-Unit

```
A: PROCEDURE OPTIONS (MAIN);
      ON FIXEDOVERFLOW BEGIN;

      END;
   CALL B;

B: PROCEDURE;
   ON UNDEFINEDFILE (PRINTFILE) OPEN
       FILE(PRINTFILE) TITLE("SYS$OUTPUT");

   CALL C:

C: PROCEDURE;

   RETURN;
   END;
```

Figure 10–3 Execution of an ON-Unit



10.4.1 Default Handling for Main Procedures

If the program was entered at a procedure with the MAIN option, PL/I searches for ON-units and performs default condition handling as follows:

1. PL/I searches for specific ON-units in the following order:

   a. A VAXCONDITION ON-unit established for the specific condition value that is being signaled

   b. A PL/I ON-unit established for a PL/I condition name, if PL/I defines a name for the condition

   c. 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 none of the above are found in any call frame, the default PL/I condition handler performs one of the following:

   - 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, its execution is attempted. 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 OpenVMS condition handler. This handler prints a message, displays a traceback, and terminates the program.

- If the signal is UNDERFLOW, a message is printed; continued execution is unpredictable, and the value that caused the condition is replaced by an undefined value. (The value would be set to zero only if OPTIONS (UNDERFLOW) were not specified on the PROCEDURE statement; and it must be specified for UNDERFLOW to be signaled.)
- If the signal is any condition other than ENDPAGE, ERROR with a fatal severity, or UNDERFLOW, 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 resignaling results in control returning to the system, the default OpenVMS 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.

10.4.2 Default Handling for Non-Main Procedures

If the call frame at which the program 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:
   a. A VAXCONDITION ON-unit established for the specific condition value that is being signaled
   b. A PL/I ON-unit established for a PL/I condition name, if PL/I defines a name for the condition
   c. 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 signaled to the caller. If the resignal results in return of control to the system, the default OpenVMS condition handler issues a message and prints a traceback. If the error was a fatal error, the default OpenVMS handler terminates the program. Otherwise, the program continues.

Example 10–4 and Figure 10–4 illustrate a search for an ON-unit that will handle a decimal overflow condition. The keyed callout numbers in the example indicate the sequence in which the search takes place.

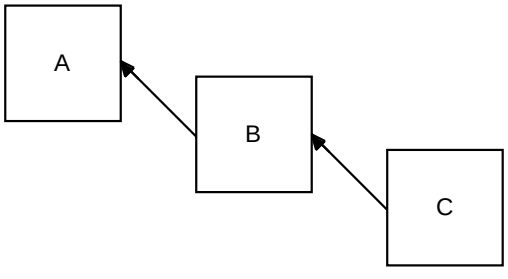
PL/I takes the following actions:

1. Searches the call frame in which the condition was signaled for three specific ON-units: VAXCONDITION(SS$_DECOV_F), FIXEDOVERFLOW, and ANYCONDITION
2. Searches the previous call frame for the same three ON-units
3. Reaches the main procedure and searches it for the three ON-units
4. Signals the ERROR condition
5. Searches the call frame in which the condition was signaled for an ERROR ON-unit
6. Searches the previous call frame for an ERROR ON-unit
7. Locates the ERROR ON-unit in the main procedure and executes it

Example 10–4 Search for an ON-Unit

```
A: PROCEDURE OPTIONS (MAIN);
   ON ERROR BEGIN; 7

   4

   END;
   CALL B;3

B: PROCEDURE; 6
   CALL C;
   RETURN; 2
   END;

C: PROCEDURE;5
   RETURN; 1
   END;
```

Figure 10–4 Search for an ON-Unit



NU–2481A–RA

10.4.3 Multiple Conditions

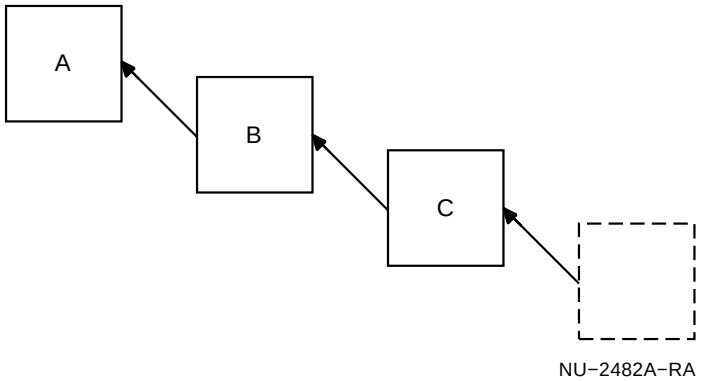
If a second condition is signaled during the execution of an ON-unit, PL/I searches for an ON-unit that will handle the second condition, beginning in the call frame in which the second condition was signaled. This handling of multiple conditions in PL/I differs from the standard behavior of the OpenVMS condition-handling facility (which skips the call frames that were searched for the current ON-unit). For information on OpenVMS condition handling, see the *Introduction to the VMS Run-Time Library*.

Example 10–5 and Figure 10–5 illustrate the search sequence followed when a second condition occurs during the execution of an ON-unit. The keyed callout numbers in the example 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. The box drawn with broken lines represents the ERROR ON-unit established by procedure B.

Example 10–5 Multiple Conditions

```
A: PROCEDURE OPTIONS (MAIN);

   1

CALL B;

B: PROCEDURE; 2
   ON ERROR BEGIN; 4

   5

   END;
   CALL C;

C: PROCEDURE; 3
   ON FIXEDOVERFLOW BEGIN;

   7
```

Figure 10–5 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 situation results when a STOP statement is executed within a FINISH or ANYCONDITION ON-unit—that is, the program will enter an infinite 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.

Note that when an ANYCONDITION ON-unit executes a nonlocal GOTO statement, the nonlocal GOTO causes an unwind, and the first ON-unit that is given control is the ANYCONDITION ON-unit itself. Thus, an infinite loop occurs. To avoid this situation, an ANYCONDITION ON-unit can contain the following lines:

```
ON ANYCONDITION BEGIN;
DECLARE SS$_UNWIND GLOBALREF VALUE FIXED;
IF ONCODE() = SS$_UNWIND THEN GOTO OKAY;
   .
   .
   .
OKAY: END;
```

This check for the condition SS$_UNWIND ensures that if a nonlocal GOTO is executed in this ON-unit, it will not cause the ON-unit to be reexecuted.

10.5 Scope of ON-Units

After an ON-unit is established, it remains in effect for the activation of the current block and all its dynamically descendent blocks, unless one of the following situations occurs:

- Another ON statement is specified for the same condition in a descendent block. The ON-unit established within the descendent block remains in effect as long as the descendent block is active.
- A REVERT statement is executed for the specified condition. A REVERT statement nullifies the most recent ON-unit for the specified condition.
- Another ON statement is specified for the same condition within the current block. Within the same block, an ON statement for a specific condition cancels the previous ON-unit.
- The block or procedure within which the ON-unit is established terminates. When a block exits, any ON-units it has established are canceled.

10.6 ON-Unit Examples

The following examples illustrate some typical ON-units. The first example establishes an ON-unit for the FINISH condition. The ON-unit ensures that two files are closed properly, and calls a routine that stops a timer in an orderly fashion.

```
ON FINISH BEGIN;
     CLOSE FILE(INFILE);
     CLOSE FILE(OUTFILE);
     CALL TIMER_END;
     END;
```

Normally, the FINISH ON-unit should be declared in the main procedure; however, it will be executed on image exit if it is established in any block that is active when that occurs.

The next example contains an ERROR ON-unit that will terminate a program in an orderly fashion, should some error occur that is not handled by a specific ON-unit.

```
DECLARE STATUS FIXED BINARY(31);
   .
   .
   .
ON ERROR BEGIN;
    CLOSE FILE (INFILE);
    CLOSE FILE (OUTFILE);
    STATUS = ONCODE();
    GOTO FINIS;
    END;
   .
   .
   .
FINIS: RETURN (STATUS);
```

The ERROR ON-unit provides a cleanup procedure to ensure that the files identified as INFILE and OUTFILE are properly closed before the image exits. The ON-unit saves the value returned by ONCODE in the variable STATUS, and transfers control to a RETURN statement that returns the numeric value to the caller. If the procedure was invoked by a RUN command, this value is returned to the command interpreter, which in turn displays on the terminal the mnemonic code for the error and the error message.

The next example contains an ON-unit that changes the value of a bit variable when end-of-file is encountered.

```
DECLARE STATE_PTR POINTER,
        STATE_FILE FILE,
        EOF BIT(1) STATIC INIT('0'B);

ON ENDFILE(STATE_FILE) EOF = '1'B;

        OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
        READ FILE(STATE_FILE) SET(STATE_PTR);
        DO WHILE (^EOF);
            .
            .
            .
            READ FILE(STATE_FILE) SET(STATE_PTR);
            END;
```

The procedure reads the records in the file STATE_FILE until it encounters end-of-file. At that point, the ON-unit executes and changes the value of EOF from 0 to 1. This action causes the test in the DO WHILE statement to fail, terminating the loop that reads the records.

Following is an example of an ON-unit that consists of a sequence of statements in a begin block.

```
ON ENDFILE (SYSIN) BEGIN;
   CLOSE FILE (TEMP);
   CALL PRINT_STATISTICS(TEMP);
   END;
```

This ON-unit consists of CLOSE and CALL statements that request special processing when the end-of-file condition occurs during reading of the default system input file, SYSIN.

Following is an example of a null statement specified for an ON-unit.

```
ON ENDPAGE(SYSPRINT);
```

This ON-unit causes PL/I to continue output on a terminal regardless of the number of lines already output. The null statement indicates that no processing is to occur when the condition occurs. Program execution continues as if the condition had been handled.

Chapter 6 demonstrates an ON-unit that handles errors encountered during record I/O operations.

10.7 Condition-Handling Built-In Functions

The following sections discuss the four PL/I for OpenVMS VAX and PL/I for OpenVMS AXP built-in functions that are useful for condition handling: ONARGSLIST, ONCODE, ONFILE, and ONKEY.

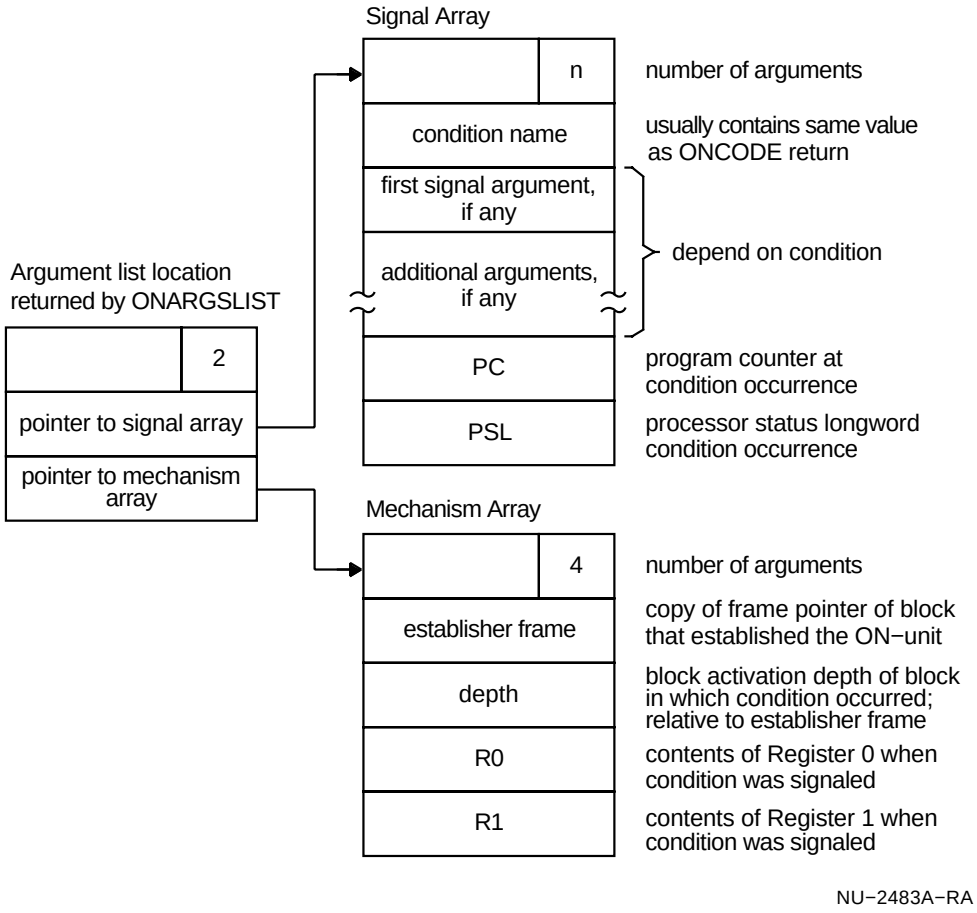
10.7.1 ONARGSLIST Built-In Function

Within the context of the PL/I language, an ON-unit is like a procedure that has no parameters. However, in the OpenVMS environment, a condition handler or ON-unit is actually called with an argument list. The argument list consists of two pointer values, each of which points to a structure containing values that provide information about the condition.

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, you can access these arguments with the ONARGSLIST built-in function; this built-in function returns a pointer to the argument list passed to the most recent ON-unit. Figure 10–6 illustrates the argument list and the arguments that can be accessed through this list.

The text module $CHFDEF contains PL/I declarations of these structures:

Figure 10–6 The Argument List Passed to an ON-Unit



NU−2483A−RA

```
/* Definitions for Signal Array and Mechanism Array arguments */
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 PLI$STARLET.TLB. You can include this module in a PL/I program by specifying the following %INCLUDE statement:

```
%INCLUDE $CHFDEF;
```

The PL/I compiler locates this module in PLI$STARLET.TLB when it compiles the source program.

Example 10–6 illustrates a procedure that displays values obtained from the signal array arguments and the mechanism array arguments. The following notes are keyed to Example 10–6:

1 The procedure includes the module $CHFDEF from the default system library.

2 The ONARGSLIST built-in function assigns a value to the pointer CHF$ARGPTR, declared in $CHFDEF.

3 Using the CHF$SIG_ARGS field in the signal array, the procedure prints the number of arguments in the signal array.

4 It displays the contents of the first argument, that is, the condition value.

5 Because the number of arguments is variable, the procedure uses the DIM built-in function to determine the number of elements in the array CHF$SIG_ARG (this array always contains three fewer members than arguments in the array, because the condition name, PC, and PSL arguments are always present).

6 After displaying the signal arguments, the procedure displays the contents of R0 and R1 from the mechanism array.

Example 10–6 Displaying Arguments Passed to a Condition Handler

```
%INCLUDE $CHFDEF;    1
DECLARE X FIXED;
           CHF$ARGPTR = ONARGSLIST();    2
/* Output number of signal arguments */
           PUT SKIP LIST('Signal Arg Count',CHF$SIG_ARGS);    3
/* Output condition name argument and rest of signal arguments */
           PUT SKIP LIST('Condition name', CHF$SIG_NAME);    4
           PUT SKIP LIST(DIM(CHF$SIG_ARG,1),
                        'additional arguments:');    5
               DO X = 1 TO DIM(CHF$SIG_ARG,1);
                   PUT SKIP LIST(CHF$SIG_ARG(X));
                   END;
/* Output RO and R1 */
           PUT SKIP(2) LIST('r0:',CHF$MCH_SAVR0);    6
           PUT SKIP(2) LIST('r1:',CHF$MCH_SAVR1);
END;
```

For more detailed information on the argument lists passed to a condition handler and for descriptions of the values in the signal array and mechanism array, see the *OpenVMS System Services Reference Manual*. Note that the PL/I run-time system signals conditions using the VAX conventions for specifying signal arguments. Specifically, it passes arguments following the requirements described for the SYS$PUTMSG procedure. This procedure is described in the *Introduction to the VMS Run-Time Library*.

10.7.2 ONCODE Built-In Function

You can use the built-in function ONCODE to obtain the specific 32-bit status value that describes any condition that is signaled. The low-order three bits of this value contain the severity of the condition (success, warning, error, or 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 10.4).

All OpenVMS-defined conditions have symbolic names associated with them. Table 10–1 lists the PL/I keyword condition names and the global symbol names for the OpenVMS 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 10–1 ONCODE Values for PL/I ON Conditions

PL/I Condition	VMS Global Symbol Name[1]
AREA	See the *PL/I for OpenVMS Systems Reference Manual* and Chapter 15 of this manual for a discussion of allocation in areas.
CONDITION	PLI$_CONDITION
CONVERSION	PLI$_CONVERSION if a SIGNAL CONVERSION statement was executed, otherwise PLI$_ONCNVPOS (PLI$_ONCNVPOS could change in a future release of PL/I)
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 or SS$_FLTOVF_F
STORAGE	The value returned by LIB$GET_VM
STRINGRANGE	PLI$_STRRANGE, or PLI$_SUBSTRn (where n is 2 or 3, indicating the 2nd or 3rd argument of the SUBSTR built-in function), or PLI$_BIFSTAPOS (indicating an out-of-range starting position for an INDEX, SEARCH, or VERIFY built-in function)
SUBSCRIPTRANGE	PLI$_SUBRANGE or PLI$_SUBRANGEn (where n is a number in the range 1 through 8 indicating the subscript number)

[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 *OpenVMS Record Management Services Reference Manual*. PL/I messages are listed in Appendix A.

(continued on next page)

Table 10–1 (Cont.) ONCODE Values for PL/I ON Conditions

PL/I Condition	VMS Global Symbol Name[1]
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$_FLTUND or SS$_FLTUND_F
VAXCONDITION	Any user-defined condition value that was signaled
ZERODIVIDE	SS$_FLTDIV, SS$_INTDIV, or SS$_FLTDIV_F

[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 *OpenVMS Record Management Services Reference Manual*. PL/I messages are listed in Appendix A.

When you write an ON-unit to handle one or more conditions, you can refer specifically to the values returned by the ONCODE built-in function using system global symbol names. Table 10–1 lists, where appropriate, the specific system global symbol for a condition name.

All symbolic names associated with OpenVMS condition values are defined as system global symbols. Thus, you can declare the names for these values in a PL/I program using the GLOBALREF and VALUE attributes, and refer to them symbolically. The linker will automatically locate the definitions of the symbols in the default system libraries when the procedure is linked.

For example, the FIXEDOVERFLOW condition can be signaled for either of two conditions. An ON-unit can determine which condition was actually signaled by testing the value of ONCODE as follows:

```
DECLARE SS$_DECOVF GLOBALREF VALUE FIXED BINARY(31);
ON FIXEDOVERFLOW BEGIN;
     IF ONCODE() = SS$_DECOVF THEN
            PUT LIST ('Decimal overflow');
         ELSE
            PUT LIST ('Integer overflow');
     END;
```

In this example, the global symbol SS$_DECOVF is declared with the GLOBALREF and VALUE attributes. The ON-unit established for the FIXEDOVERFLOW condition determines, from the value returned by ONCODE, whether the condition was specifically a decimal overflow or an integer overflow.

10.7.3 ONFILE Built-In Function

The ONFILE built-in function returns the name of the file constant for which the current file-related condition was signaled. Its format is given in the *PL/I for OpenVMS Systems Reference Manual*.

This built-in function can be used in any of the following ON-units:

- An ON-unit established for the KEY, ENDFILE, ENDPAGE, and UNDEFINEDFILE conditions
- A VAXCONDITION ON-unit established for I/O errors that can occur during file processing

- An ERROR ON-unit that receives control as a result of the default PL/I action for file-related errors, which is to signal the ERROR condition

The returned value is a varying-length character string. If referenced outside an ON-unit or within an ON-unit that is executed as a result of a SIGNAL statement, the ONFILE function returns a null string.

10.7.4 ONKEY Built-In Function

The ONKEY built-in function returns the key value that caused the KEY condition to be signaled during an I/O operation to a file that is being accessed by key. Its format is given in the *PL/I for OpenVMS Systems Reference Manual*.

This built-in function can be used in an ON-unit established for these conditions:

- The KEY, ENDFILE, or UNDEFINEDFILE conditions
- An ERROR ON-unit that receives control as a result of the default PL/I action for the KEY condition, which is to signal the ERROR condition

The returned key value is a varying-length character string. If referenced outside an ON-unit, or within an ON-unit executed as a result of the SIGNAL statement, the ONKEY built-in function returns a null string.

11

Using PL/I in the Common Language Environment

The PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compilers are part of the OpenVMS common language environment. This environment defines certain calling procedures and guidelines that allow you to call routines written in different languages or prewritten system routines from PL/I for OpenVMS VAX and PL/I for OpenVMS AXP. You can call any of the following routine types from PL/I for OpenVMS VAX and PL/I for OpenVMS AXP:

- Routines written in other OpenVMS VAX and OpenVMS AXP languages
- OpenVMS Run-Time Library routines
- OpenVMS system services
- OpenVMS utility routines

The terms routine, procedure, and function are used throughout this chapter.

Note

The definitions of *routine*, *procedure*, and *function* used throughout this chapter are somewhat different from standard PL/I for OpenVMS VAX and PL/I for OpenVMS AXP terminology. These definitions are used for consistency with the OpenVMS system routines documentation and apply only throughout this chapter.

A *routine* is a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and optionally an argument list. Procedures and functions are specific types of routines: a *procedure* is a routine that does not return a value, whereas a *function* is a routine that returns a value by assigning that value to the function's identifier.

System routines are prewritten OpenVMS routines that perform common tasks, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program, provided that PL/I supports the data structures required to call the routine. The system routines used most often are OpenVMS Run-Time Library routines, utility routines, and system services. System routines, which are discussed later in this chapter, are documented in detail in the *VMS Run-Time Library Routines Volume*, the *OpenVMS Utility Routines Manual*, and the *OpenVMS System Services Reference Manual*.

11.1 OpenVMS Calling Standard

The *OpenVMS Calling Standard* describes the concepts used by OpenVMS VAX and OpenVMS AXP languages for invoking routines and passing data between them. It also describes the differences between the VAX and AXP parameter passing mechanisms. The *OpenVMS Calling Standard* specifies the following attributes:

- Register usage
- Stack usage
- Function value return
- Argument list

The following sections discuss these attributes in more detail for OpenVMS VAX systems. For more detail on OpenVMS AXP systems, see the *OpenVMS Calling Standard*.

The *OpenVMS Calling Standard* also defines such attributes as the calling sequence, the argument data types and descriptor formats, condition handling, and stack unwinding. These attributes are discussed in detail in the *Introduction to VMS System Routines*.

11.1.1 Register and Stack Usage

The *OpenVMS Calling Standard* defines registers and their uses, as listed in Table 11–1 and Table 11–2.

Table 11–1 VAX Register Usage

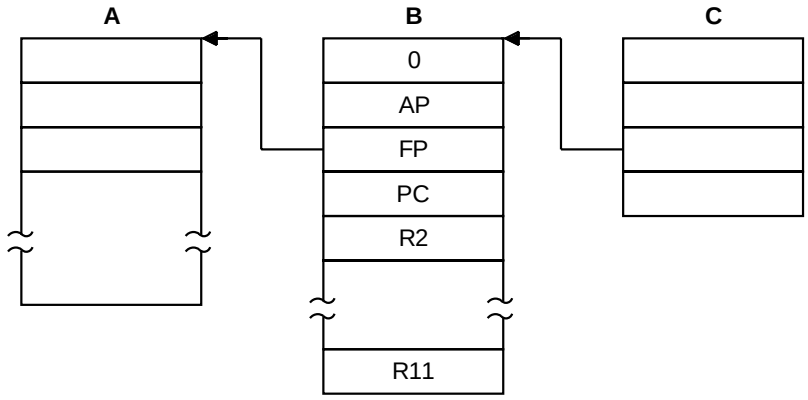
Register	Use
PC	Program counter
SP	Stack pointer
FP	Current stack frame pointer
AP	Argument pointer (when necessary)
R1	Environment value (when necessary)
R0, R1	Function value return registers

Table 11–2 AXP Register Usage

Register	Use
PC	Program counter
SP	Stack pointer
FP	Frame pointer for current procedure
R25	Argument information register
R16 to R21, F16 to F21	Argument list registers
R0	Function value return register

By definition, any called routine can use registers R2 through R11 for computation, and the AP register as a temporary register.

Figure 11–1 The Call Stack



Legend

AP − copy of argument pointer for procedure A
FP − pointer to A's call frame
PC − memory location in A at which B was invoked
R2−R11 − contents of A's general registers R2 through R11

NU−2463A−RA

In the *OpenVMS Calling Standard*, a *stack* is defined as a LIFO (last-in/first-out) temporary storage area that the system allocates for every user process. The system keeps information about each routine call in the current image on the call stack. Then, each time you call a routine, the system creates a structure on this call stack, known as the *call frame*. The call frame for each active procedure contains the following:

- A pointer to the call frame of the previous routine call. This pointer corresponds to the frame pointer (FP).
- The argument pointer (AP) of the previous routine call.
- The storage address of the point at which the routine should return; that is, the address of the instruction following the call to the current routine. This address is called the program counter (PC).
- The contents of other general registers. Based on a mask specified in the control information, the system restores the saved contents of these registers to the calling routine when control returns to it.

Figure 11–1 illustrates the call stack and several call frames. Procedure A calls procedure B, which calls procedure C.

When a routine completes execution, the system uses the frame pointer in the call frame of the current routine to locate the frame of the previous routine. The system then removes the call frame of the current routine from the stack.

11.1.2 Return of the Function Value

A function is a routine that returns a single value to the calling routine. The *function value* is the value returned to the caller. According to the calling standard, a function value may be returned as either an actual value or a condition value that indicates success or failure.

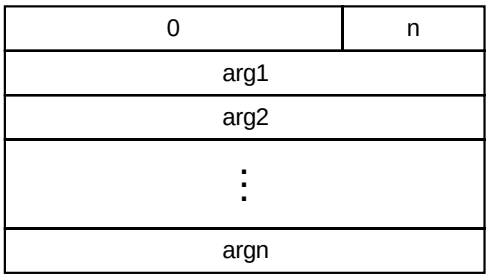
11.1.3 The Argument List

The *OpenVMS Calling Standard* also defines a data structure called the argument list. You use an argument list to pass information to a routine and receive results.

On AXP systems, an argument list is formed using registers R16 to R21 or F16 to F21 and a collection of quadwords in memory (depending on the number and type of the arguments.

On VAX systems, an *argument list* is a collection of longwords in memory that represents a routine parameter list and possibly includes information for the return of a function value, if a function returns a value that is not suitable for return in R0 or R0/R1. Figure 11–2 shows the structure of a typical OpenVMS VAX argument list.

Figure 11–2 Structure of an OpenVMS VAX Argument List



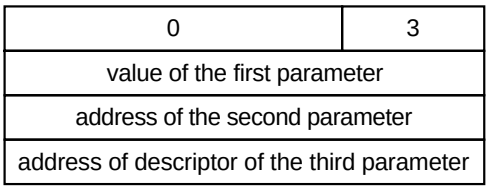
NU−2464A−RA

The first longword must be present; this longword stores the number of arguments (the argument count: n) as an unsigned integer value in the low byte of the longword. The remaining 24 bits of the first longword are reserved for use by Digital and should be zero. The longwords labeled arg1 through argn are the actual parameters, which can be any of the following:

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

The argument list contains the parameters that are passed to the routine. Depending on the passing mechanisms for these parameters, the forms of the arguments contained in the argument list vary. For example, if you pass three arguments, the first by value, the second by reference, and the third by descriptor, the argument list contains the value of the first argument, the address of the second, and the address of the descriptor of the third. Figure 11–3 shows this argument list.

Figure 11–3 Example of an OpenVMS VAX Argument List



NU–2465A–RA

11.2 Parameter-Passing Mechanisms

To pass data between routines that are not written in the same VAX or AXP language, you must specify how you want that data to be represented and interpreted. You do this by specifying a *parameter-passing mechanism*. The three general parameter-passing mechanisms and their keywords and abbreviations in PL/I are as follows:

- By reference—REFERENCE (REF)
- By descriptor—DESCRIPTOR (DESC)
- By value—VALUE (VAL)

The parameter-passing mechanisms are intended for use only in calling non-PL/I routines. External routines written in PL/I should be called with the default mechanisms.

The following sections describe the arguments in terms of PL/I data types, dummy arguments created (if any), parameter-passing mechanism conventions, and attributes to define the manner in which parameters are to be passed. 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.

11.2.1 Passing Parameters by Reference

When you pass a parameter by reference, the PL/I for OpenVMS VAX or PL/I for OpenVMS AXP compiler passes the address at which the actual parameter value is stored. In other words, your routine has access to the parameter's storage address. Therefore, when you manipulate and change the value of this parameter, the changed value overwrites the previous value of the parameter. Thus, when you pass a parameter by reference, any changes that you make to the value of the parameter in your routine are reflected in the calling routine as well, provided a dummy argument is not created.

You can pass an argument of any data type by reference. This mechanism is the default used by PL/I for OpenVMS VAX and PL/I for OpenVMS AXP for all arguments except character strings or arrays with nonconstant extents, and unaligned bit strings. Note that bit-string variables must have the ALIGNED attribute to be passed by reference. In general, 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 value and its second argument to be passed by reference. This PL/I for OpenVMS VAX procedure can be declared as follows:

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

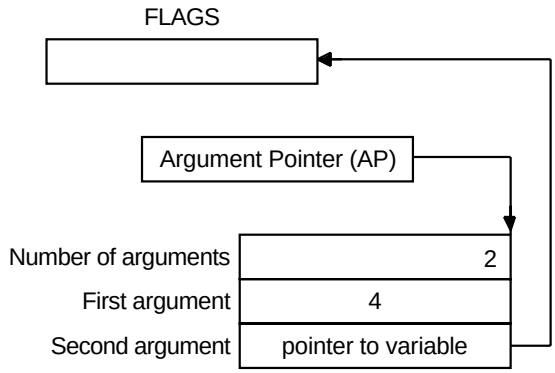
Note

Most OpenVMS routines have declarations in PLI$STARLET; it is therefore not normally necessary to declare them yourself.

When this procedure is invoked, the second argument must be a variable declared as BIT(32) ALIGNED for PL/I for OpenVMS VAX or BIT(64) ALIGNED for PL/I for OpenVMS AXP. PL/I passes the argument by reference. The following PL/I for OpenVMS VAX code example with Figure 11–4 illustrates argument passing by reference.

```
DECLARE FLAGS BIT(32) ALIGNED;
DECLARE SYS$READEF ENTRY (
        FIXED BINARY(31) VALUE,
        BIT(32) ALIGNED;
     .
     .
     .
CALL SYS$READEF(4, FLAGS);
```

Figure 11–4 Argument Passing by Reference



NU–2466A–RA

The data types in the parameter descriptors of all output arguments must match the data types of the written arguments.

11.2.1.1 Using the ANY Attribute

When you write a parameter descriptor for a non-PL/I procedure, you can specify the ANY attribute without the VALUE attribute to describe an argument that is to be passed by reference. The argument can be of any addressable data type known to PL/I for OpenVMS VAX or PL/I for OpenVMS AXP. For example, PL/I for OpenVMS VAX, the SYS$READEF service can be specified as follows:

```
DECLARE SYS$READEF ENTRY (FIXED BINARY(31) VALUE, ANY);
```

The second parameter descriptor in the ENTRY attribute indicates that the second argument is to be passed by reference to the procedure SYS$READEF and that it can have any data type. When you specify ANY for an argument to be passed by reference, you cannot specify data type attributes. Note that if you specify the VALUE attribute in conjunction with the ANY attribute, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP pass the argument by value.

The ANY attribute is especially useful when you must specify a data structure as an argument. You need not declare the structure within the parameter descriptor, only the ANY attribute.

11.2.1.2 Dummy Arguments for Arguments Passed by Reference

When PL/I for OpenVMS VAX or PL/I for OpenVMS AXP passes an argument by reference, it places either the address of the actual argument or the address of a dummy argument in the argument list of the called procedure. PL/I for OpenVMS VAX and PL/I for OpenVMS AXP create a dummy argument in the following cases:

- When the written argument is a constant or an expression
- When the written argument is enclosed in parentheses
- When the data type of the written argument does not match the data type or precision specified in the parameter descriptor

In the last case listed above, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP issue an informational or warning message and, for scalar arguments, creates a dummy argument of the data type of the parameter. It places the address of this dummy argument in the argument list. If the argument is an aggregate, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP issue an error message; it does not create a dummy argument for an array or for a structure.

In creating a dummy argument, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP perform the following conversions:

Data Type of Written Argument	Data Type of Dummy Argument
BIT (unaligned)	BIT ALIGNED
FIXED BINARY (p,0) or FIXED DECIMAL (p,0) (VAX)	FIXED BINARY (31)
FIXED BINARY (p,0) or FIXED DECIMAL (p,0) (AXP)	FIXED BINARY (63)
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.

11.2.1.3 Using Pointer Values for Arguments Passed by Reference

When an argument is passed by reference, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP place 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:

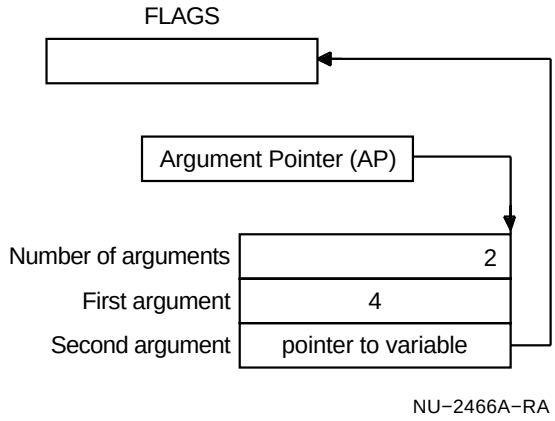
```
DECLARE SYS$READEF (ANY VALUE, POINTER VALUE),
        FLAGS BIT(32) ALIGNED;
CALL SYS$READEF (4, ADDR(FLAGS));
```

At this procedure invocation, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP place the the pointer value returned by the ADDR built-in function directly in the argument list.

Figure 11–5 illustrates the argument list for the following example. Note that the actual argument list in this example corresponds to the argument list shown previously in Figure 11–4.

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

Figure 11–5 Passing a Pointer Value as an Argument



NU–2466A–RA

11.2.1.4 Passing Arrays to a FORTRAN Routine by Reference

In FORTRAN, arrays must always be passed by reference; the array's extents are, by custom, passed as separate arguments. The REFERENCE attribute provides a convenient way to express an array parameter for FORTRAN. For example:

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

DECLARE (S, X(*)) FLOAT;

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

SUM is a FORTRAN routine that sums the elements of a one-dimensional array of floating-point numbers. Its second parameter is the number of elements in the array.

11.2.2 Passing Parameters by Descriptor

When you pass a parameter by descriptor, the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compiler pass the address of a descriptor to the called routine. A *descriptor* is a data structure that contains the address of a parameter, along with other information such as the parameter's data type and size.

For some structure parameters, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP pass an abbreviated form of descriptor that contains only essential position and extent information. In these cases, the address of the abbreviated descriptor is placed in the argument list for the called routine. The use of an abbreviated descriptor is transparent to you.

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP normally pass parameters by descriptor in the following cases:

- When a parameter descriptor specifies a character string with an asterisk length or an array with asterisk extents
- When a parameter descriptor specifies an unaligned bit string or an array or structure consisting entirely of unaligned bit strings
- When a parameter descriptor specifies a structure containing any strings or arrays with asterisk extents

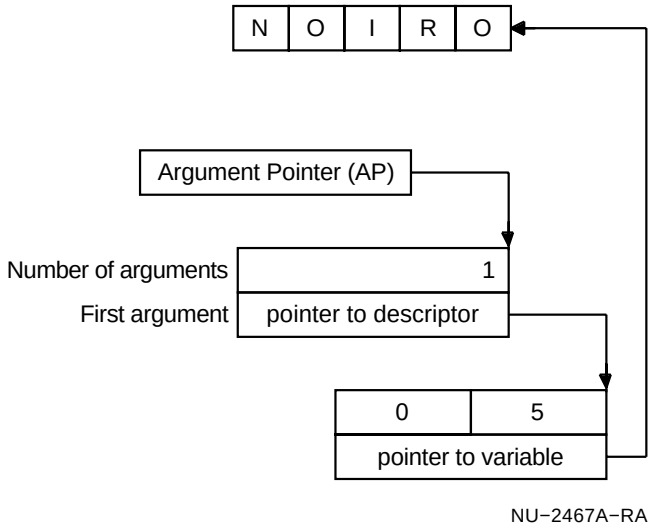
For example, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP pass the arguments associated with the following parameter descriptors by descriptor:

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

Figure 11–6 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 descriptors.

```
DECLARE NAME CHARACTER(5);
       STATIC INITIAL ('ORION');
DECLARE SYS$SETPRN ENTRY
       (CHARACTER(*));
     .
     .
     .
CALL SYS$SETPRN(NAME);
```

Figure 11–6 Argument Passing by Descriptor



11.2.2.1 Passing Character Strings

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

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

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

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

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

11.2.2.2 Passing Varying Character Strings

If you specify a varying character-string argument for a parameter declared as (CHARACTER(*), the PL/I for OpenVMS VAX or PL/I for OpenVMS AXP compiler issues a warning message, constructs a fixed-length character-string dummy argument, and creates a character-string descriptor for the dummy argument.

For an input argument, as in the example of the SYS$SETPRN service, the dummy argument is acceptable. To suppress the warning message during compilation, enclose the argument in parentheses. For example, if NAME is a variable declared with the CHARACTER VARYING attributes, you can specify it as an argument to the SYS$SETPRN system service like this:

```
CALL SYS$SETPRN ((NAME));
```

The parentheses around the argument NAME force PL/I for OpenVMS VAX and PL/I for OpenVMS AXP to create a dummy argument. The compiler does not issue a warning about a nonmatching parameter and argument.

For a non-PL/I routine that returns a character string to a variable, however, you cannot use a varying character string for an argument. If the actual output argument is declared as VARYING and the parameter descriptor specifies CHARACTER(*), PL/I for OpenVMS VAX and PL/I for OpenVMS AXP create a dummy argument and the actual argument is not modified. Thus, for output character strings passed by character-string descriptor, you must choose one of the following:

- Specify a fixed-length character-string variable to receive the string and a fixed-point binary variable to receive the length of the string.
- Construct an actual character-string descriptor and pass the name of the character-string descriptor as an argument. This technique is described in Section 11.2.2.5.

In either case, you must include in your program the statements necessary to determine the length of the string returned. For example, the SYS$ASCTIM system service returns a character-string time value to a character-string descriptor and returns the length of the string to a fixed-point binary variable. These two arguments can be declared as follows:

```
DECLARE TIME CHARACTER(63),
        TIME_LENGTH FIXED BINARY(15);
```

After the call to this procedure, the following statement might output the equivalence name returned:

```
PUT LIST ('Time is ',SUBSTR(TIME,1,TIME_LENGTH));
```

The PUT statement uses the SUBSTR built-in function to obtain the length of the string returned in the variable TIME_LENGTH by SYS$ASCTIM.

11.2.2.3 Using ANY CHARACTER(*)

You can use the ANY CHARACTER(*) declaration to declare parameters for routines that can handle both fixed- or varying-length strings for a single parameter. The declaration allows either VARYING or NONVARYING strings to be passed without the creation of a dummy argument. Essentially, all OpenVMS system routines, with the exception of the system services and Librarian Utility routines, can accept string parameters of this type.

Note that routines written in other languages do not allow strings to be passed using this method by default. The routine being called must explicitly use LIB$ANALYZE_SDESC (or an equivalent routine) for this declaration to work correctly.

You can use the ANY CHARACTER(*) attribute as shown in the following example:

```
%INCLUDE $STSDEF;
DECLARE FIXED_STRING CHAR(22),
        VARYING_STRING CHAR(80) VARYING;

DECLARE LIB$DATE_TIME ENTRY(
            ANY CHARACTER(*))
            RETURNS (FIXED BINARY(31));

STS$VALUE = LIB$DATE_TIME(FIXED_STRING);
STS$VALUE = LIB$DATE_TIME(VARYING_STRING);
```

In both of these cases, the string contains the output value, since no dummy argument is required.

11.2.2.4 Using ANY DESCRIPTOR

The ANY and DESCRIPTOR attributes can be used together for routines that can process any valid data type descriptor. Routines of this type are few; however, LIB$CVT_DX_DX and the routine DTR$COMMAND in the VAX DATATRIEVE layered product callable interface allow any valid data type descriptor. For these routines, declarations should be as follows:

```
DECLARE LIB$CVT_DX_DX ENTRY(
                ANY DESCRIPTOR,
                ANY DESCRIPTOR,
                FIXED BINARY(15) OPTIONAL TRUNCATE)
              RETURNS(FIXED BINARY(31));

DECLARE DTR$COMMAND ENTRY(
                1 LIKE DTR_ACCESS_BLOCK,
                ANY CHARACTER(*),
                ANY DESCRIPTOR LIST TRUNCATE);
```

11.2.2.5 Passing an Actual Descriptor

To pass an actual descriptor as an argument, you must take the following steps; the keyed numbers correspond to the callout numbers in Example 11–1.

1. In the parameter descriptor for the called procedure, declare a structure in the format of a descriptor for the argument that is to be passed by descriptor, specify ANY in the parameter descriptor, or use the REFERENCE built-in function to override the parameter declaration at the point of the call.
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 system service (SYS$SETPRN) shown in Example 11–1 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 11–6 of passing an argument by descriptor.

Example 11–1 Writing a Character-String Descriptor

```
DECLARE SYS$SETPRN ENTRY (CHARACTER(*));1

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

DECLARE NEWNAME CHARACTER (5) STATIC INITIAL ('ORION');
NAME_DESC.NAME_LENGTH = LENGTH(NEWNAME);3
NAME_DESC.NAME_ADDRESS = ADDR(NEWNAME);3

CALL SYS$SETPRN(REF(NAME_DESC));4
```

Note that this example can be simplified if SYS$SETPRN is declared as follows:

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

All other variables would be the same as in Example 11–1, although the use of the REFERENCE built-in function could be omitted.

In most cases, the first method is preferable because it allows the declarations in PLI$STARLET to be used consistently; the only cases in which special handling is required is for those calls that require it.

11.2.3 Passing Parameters by Value

When you pass a parameter by value, the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compiler pass a copy of the parameter's value to the routine instead of passing its address. Because only a copy of the parameter's value is passed, the routine does not have access to the storage location of the parameter. Therefore, when you pass a parameter by value, any changes that you make to the parameter value in the called routine do not affect the value of that parameter in the calling routine.

For an argument to be passed by value, you must use the VALUE attribute in a parameter description. The following declaration of the external entry VHF illustrates a declaration for an external routine that receives its parameter by value.

```
DECLARE VHF ENTRY (FIXED BINARY(31) VALUE);
```

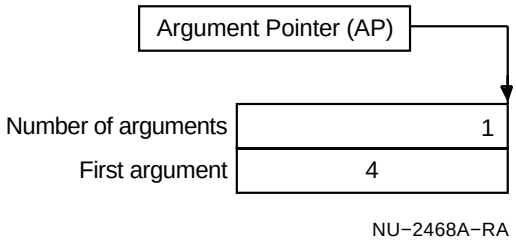
You can also define PL/I for OpenVMS VAX and PL/I for OpenVMS AXP procedures that receive arguments by value. To do this, you must specify the VALUE attribute in the declaration of the parameter. For example, the corresponding definition of the routine VHF would be as follows:

```
VHF PROCEDURE (LENGTH);
   .
   .
   .
DECLARE LENGTH FIXED BINARY(31) VALUE;
```

The following code example and Figure 11–7 illustrate argument passing by value.

```
DECLARE VHF ENTRY(
        FIXED BINARY(31) ANY VALUE);
    .
    .
    .
CALL VHF(4);
```

Figure 11–7 Argument Passing by Immediate Value



Arguments that can be passed by value are limited to the following data types, which can be expressed in 32 bits for OpenVMS VAX and 64 bits for OpenVMS AXP:

- OpenVMS VAX
  - FIXED BINARY(m), where m is less than or equal to 31
  - FLOAT BINARY(n), where n is less than or equal to 24
  - BIT(p) or BIT(p) ALIGNED, where p is less than or equal to 32
  - ENTRY
  - OFFSET
  - POINTER
- OpenVMS AXP
  - FIXED BINARY(m), where m is less than or equal to 63
  - FLOAT BINARY(n), where n is less than or equal to 53
  - BIT(p) or BIT(p) ALIGNED, where p is less than or equal to 64
  - ENTRY
  - OFFSET
  - POINTER

No other data types can be passed by value. Note that when ENTRY VALUE is specified in a parameter descriptor, only the entry points of external routines may be passed by value. A complete entry value for an internal routine requires two longwords, one for the parent frame and one for the 32-bit entry-point address.

When you specify the VALUE attribute in a parameter descriptor, you can specify the ANY attribute instead of declaring any data type attributes. For example, the declaration of VHF can appear as follows:

```
DECLARE VHF ENTRY (ANY VALUE);
```

At the time of the procedure's invocation, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP convert the written argument as needed to create a longword dummy argument.

11.2.3.1 Dummy Arguments for Arguments Passed by Value

For arguments to be passed by value, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP always create a dummy argument directly in the argument list for the called procedure. If the parameter descriptor is specified with the ANY and VALUE attributes, dummy arguments are created with the following data types:

Data Type of Written Argument	Data Type of Dummy Argument
FIXED BINARY, (p,0) or FIXED DECIMAL (p,0)	FIXED BINARY (31)for OpenVMS VAX or FIXED BINARY (63) for OpenVMS AXP
BIT or BIT ALIGNED	BIT (32) ALIGNED for OpenVMS VAX or BIT (64) ALIGNED for OpenVMS AXP
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.

11.2.4 Special Parameter Attributes

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP provide the LIST, OPTIONAL, and TRUNCATE parameter attributes.

- The LIST attribute indicates that the specified list can be invoked with a list of arguments for the last parameter.
- The OPTIONAL attribute indicates that optional arguments will not be specified in the entry invocation.
- The TRUNCATE attribute indicates that the argument list may be truncated at a specified position.

Each of these attributes is described in the following sections.

11.2.4.1 LIST Attribute

Although most system routines and procedures require a specific number of arguments, some routines accept an unspecified number of optional arguments. To declare these routines in a PL/I for OpenVMS VAX or PL/I for OpenVMS AXP program so that you can invoke them with differing numbers of arguments, you must declare the last parameter with the LIST attribute. The last parameter descriptor given in the ENTRY attribute is used for extra arguments.

The Formatted ASCII Output system service (SYS$FAO) 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) OPTIONAL,
                       CHAR(*), ANY VALUE LIST TRUNCATE);
```

This parameter descriptor specifies only four arguments. When SYS$FAO is invoked with more than four arguments, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP use 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.

Note

The LIST attribute is valid only for parameter descriptors.

11.2.4.2 OPTIONAL Attribute

Some PL/I for OpenVMS VAX, PL/I for OpenVMS AXP, and non-PL/I routines with fixed-length argument lists accept optional arguments and provide a default action if no value is specified for the optional argument. When an optional argument is not specified, its corresponding argument list longword is filled with a value of zero passed by immediate value.

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, 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:

```
(,,)
```

The called procedure must detect and interpret the optional parameters in the argument list. The following example illustrates optional arguments omitted from an argument list:

```
DECLARE SYS$ASCTIM ENTRY (
                      FIXED BINARY(15) OPTIONAL,
                      CHARACTER(*),
                      ANY OPTIONAL,
                      BIT(1) ALIGNED VALUE OPTIONAL),
        TIME_STRING CHARACTER(24);
   .
   .
   .
CALL SYS$ASCTIM(,TIME_STRING,,);
```

This call to the service SYS$ASCTIM specifies only the second argument; the argument list contains four arguments. When you specify a null argument, as above, PL/I for OpenVMS VAX and PL/I for OpenVMS AXP always place a zero in the argument list passed to the called procedure.

11.2.4.3 TRUNCATE Attribute

Some routines may allow an argument list to be truncated in a specified location. For example:

```
DECLARE LIB$GET_INPUT ENTRY (ANY CHARACTER(*),
                          ANY CHARACTER(*) OPTIONAL TRUNCATE,
                          FIXED BINARY(15) OPTIONAL TRUNCATE);
```

This declaration of the Get Line from SYS$INPUT routine (LIB$GET_INPUT) specifies that the argument list may vary in length. In fact, LIB$GET_INPUT has three parameters, but the last two parameters are optional.

When you invoke such a routine omitting trailing arguments, you do not have to account for all the arguments in the procedure invocation argument list. For example, the LIB$GET_INPUT routine can be invoked with trailing arguments omitted:

```
CALL LIB$GET_INPUT(GET_STRING);
```

The variable GET_STRING is specified for the first argument. The other two arguments are not specified.

11.2.5 Summary of Rules for Passing Parameters

You can specify the passing mechanism—reference, value, or descriptor—for a parameter in three ways:

- Use an attribute.
- Use a built-in function.
- Use a default.

The following rules and examples explain these three alternatives.

- Specify the REFERENCE, VALUE, or DESCRIPTOR attribute explicitly when you declare the parameter. For example:

  ```
  CHAR(*) REFERENCE
  (*) FIXED BINARY REFERENCE
  FIXED BIN(31) VALUE
  BIT(32) ALIGNED VALUE
  FLOAT BIN(23) VALUE
  ENTRY (FIXED BIN) RETURNS(FIXED BIN) VALUE
  FIXED BINARY DESCRIPTOR
  FIXED DECIMAL(10,2) DESCRIPTOR
  CHAR(10) DESCRIPTOR
  ```

- Use the REFERENCE, VALUE, or DESCRIPTOR built-in function at the call, overriding whatever passing mechanism would otherwise be used. For example:

  ```
  CALL E (REF(10));
  CALL FOO (REF(E));
  CALL I (VALUE(10));
  CALL FOO (VALUE(E));
  CALL E (DESC(10));
  CALL FOO (DESC(E));
  ```

  The built-in function completely overrides the parameter declaration and consequently no type conversion is performed. The built-in function evaluates a numeric constant as FIXED BINARY(31,0) rather than FIXED DECIMAL. For example, REF(10) passes a longword containing the binary value 10 by reference.

- Use the default passing mechanism for the data type of the parameter. By default, parameters are passed by reference except for unaligned bit strings and parameters containing asterisk (*) extents, which are passed by descriptor. (Value is never the default passing mechanism.) For example, parameters of the following types are all passed by reference:

```
CHAR(10) VARYING
BIT(22) ALIGNED
(10) FIXED DEC
(20,-1:7) FLOAT
CHAR
1,2 (10) FIXED, 2 (8) FLOAT
```

Parameters of the following types are passed by descriptor, by default, because they contain asterisk (*) extents or an unaligned bit string:

```
CHAR(*)
BIT(10) UNALIGNED
(*) FIXED BIN
1,2(*) FIXED DEC, 2(10) FLOAT
```

Restrictions

For passing by reference, a parameter must be addressable. Thus, entry constants, file constants, and unaligned bit strings cannot be passed by reference.

For passing by value, a parameter cannot occupy more than 32 bits for OpenVMS VAX or 64 bits for OpenVMS AXP. and must be one of the following types:

- OpenVMS VAX
  - FIXED BINARY(m), where m is less than or equal to 31
  - FLOAT BINARY(n), where n is less than or equal to 24
  - BIT(p) or BIT(p) ALIGNED, where p is less than or equal to 32
  - ENTRY
  - OFFSET
  - POINTER
- OpenVMS AXP
  - FIXED BINARY(m), where m is less than or equal to 63
  - FLOAT BINARY(n), where n is less than or equal to 53
  - BIT(p) or BIT(p) ALIGNED, where p is less than or equal to 64
  - ENTRY
  - OFFSET
  - POINTER

11.3 OpenVMS Run-Time Library Routines

The OpenVMS Run-Time Library is a library of prewritten, commonly used routines that perform a wide variety of functions. These routines are grouped according to the types of tasks they perform, and each group has a prefix that identifies those routines as members of a particular OpenVMS Run-Time Library facility. Table 11–3 lists all the language-independent run-time library facility prefixes and the types of tasks each facility performs.

Table 11–3 Run-Time Library Facilities

Facility Prefix	Types of Tasks Performed
DTK$	DECtalk routines that are used to control Digital's DECtalk device
LIB$	Library routines that obtain records from devices, manipulate strings, convert data types for I/O, allocate resources, obtain system information, signal exceptions, establish condition handlers, enable detection of hardware exceptions, and process cross-reference data
MTH$	Mathematics routines that perform arithmetic, algebraic, and trigonometric calculations
OTS$	General purpose routines that perform tasks such as data type conversions as part of a compiler's generated code
SMG$	Screen management routines that are used in designing, composing, and keeping track of complex images on a video screen
STR$	String manipulation routines that perform such tasks as searching for substrings, concatenating strings, and prefixing and appending strings

The file SYS$LIBRARY:PLI$STARLET.TLB defines the entry points for these routines.

11.4 OpenVMS System Service Routines

System services are prewritten system routines that perform a variety of tasks, such as controlling processes, communicating among processes, and coordinating I/O.

Unlike the OpenVMS Run-Time Library routines, which are divided into groups by facility, all system services share the same facility prefix (SYS$). These system services are logically divided into groups that perform similar tasks. Table 11–4 describes these groups.

Table 11–4 System Services

Group	Types of Tasks Performed
AST	Allows processes to control the handling of ASTs
Change Mode	Changes the access mode of particular routines
Condition Handling	Designates condition handlers for special purposes
Event Flag	Clears, sets, reads, and waits for event flags, and associates with event flag clusters
Information	Returns information about the system, queues, jobs, processes, locks, and devices
Input/Output	Performs I/O directly, without going through VAX RMS
Lock Management	Enables processes to coordinate access to shareable system resources
Logical Names	Provides methods of accessing and maintaining pairs of character string logical names and equivalence names

(continued on next page)

Table 11–4 (Cont.) System Services

Group	Types of Tasks Performed
Memory Management	Increases or decreases available virtual memory, controls paging and swapping, and creates and accesses shareable files of code or data
Process Control	Creates, deletes, and controls execution of processes
Security	Enhances the security of OpenVMS systems
Time and Timing	Schedules events, and obtains and formats binary time values

The file SYS$LIBRARY:PLI$STARLET.TLB defines the entry points for these routines.

11.5 OpenVMS Utility Routines

Utility routines are prewritten system routines that you can use to perform a variety of tasks such as sorting and library maintenance. Table 11–5 lists some of the commonly used VMS utilities.

Table 11–5 VMS Utilities

Utility Prefix	Types of Tasks Performed
ACLEDIT$	Callable interface to the Access Control List Editor
CLI$	Command line parsing
LBR$	Library manipulation
PSM$	Print Symbiont modification
SMB$	Job controller and symbiont process interface
SOR$	Sort/Merge procedures
TPU$	Callable interface to the VAX Text Processing Utility

The file SYS$LIBRARY:PLI$STARLET.TLB defines all of the entry points for these routines.

11.5.1 OpenVMS SORT/MERGE Routines

The OpenVMS SORT Utility (SORT) 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 for OpenVMS VAX or PL/I for OpenVMS AXP program, you can call SORT routines.

The SORT routines can be used in either of two sequences:

- You can sort all of the records in an entire file. This process is similar to calling SORT from the DCL command level to sort a file.
- You can process the records in a file, pass them to SORT one at a time, and request SORT to merge and sort the records and then to return them one at a time. This process allows you to perform some intermediate operation on each record before initiating the actual sorting of the records.

Examples of calling SORT routines are included in Section 11.8 of this chapter.

11.6 Calling Routines

The basic steps for calling routines are the same whether you are calling a routine written in PL/I for OpenVMS VAX or PL/I for OpenVMS AXP, a routine written in some other OpenVMS language, a system service, a utility routine, or an OpenVMS Run-Time Library routine. The following sections outline the procedures for calling non-PL/I routines.

11.6.1 Determining the Type of Call

Before you call an external routine, you must first determine whether the call should be a procedure call or a function call. You should call a routine as a procedure if it does not return a value. You should call a routine as a function if it returns any type of value.

11.6.2 Declaring an External Routine and Its Arguments

To call a routine written in another OpenVMS language, or to call a system routine, you need to declare the routine as an external procedure or function and to declare the data types and passing mechanisms of its arguments. Note that arguments can be either required or optional.

You should include the following information in an external routine declaration:

- The name of the routine
- The data types of all the routine parameters
- The passing mechanisms for all the routine parameters, provided that the routine is not written in PL/I for OpenVMS VAX or PL/I for OpenVMS AXP

11.6.3 Calling the External Routine

Once you have declared a routine, you can call it. To call a procedure, you use the CALL statement. To call a function, you can use a function invocation either in an assignment statement or as an argument in another routine call. In either case, you must specify the name of the routine being called and all parameters required for that routine. Make sure the data types and passing mechanisms for the actual parameters you are passing coincide with those you declared earlier, and with those declared in the routine.

If you do not want to specify a value for a required parameter, you can pass a null argument by inserting a comma (,) as a placeholder in the argument list. If you use any passing mechanism other than the default, you must specify the passing mechanism in the CALL statement or the function call.

At this point, the routine being called receives control, executes, and then returns control to the calling routine at the next statement after the CALL statement or function call.

11.6.4 Calling System Routines

The basic steps for calling system routines are the same as those for calling any external routine. However, when calling system routines, you need to provide some additional information that is discussed in the following sections.

11.6.4.1 Declaring System Routines

The default PL/I for OpenVMS VAX and PL/I for OpenVMS AXP text library PLI$STARLET.TLB contains declarations for all the system routines as external entries. The text module names are most often the same as the routine entry points. 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$TRNLNM;
```

The PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compilers, by default, locate the module SYS$TRNLNM in PLI$STARLET.TLB during compilation.

In PLI$STARLET.TLB, any system service that has a wait for event flag version is found in the same module as the version without the wait. Thus, to get the definition of any wait ("W") form, you need to use the base (nonwait) form of the service in the %INCLUDE statement. For example, to get the definition for SYS$GETJPIW, you should use the following:

```
%INCLUDE SYS$GETJPI;
```

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 for OpenVMS VAX or PL/I for OpenVMS AXP program, the linker searches this library by default.

11.6.4.2 System Routine Arguments

All the system routine arguments are described in terms of the following information:

- OpenVMS usage
- Data type
- Type of access allowed
- Passing mechanism

OpenVMS usages are data structures that are layered on the standard OpenVMS data types. For example, the OpenVMS usage mask_longword signifies an unsigned longword integer that is used as a bit mask, and the OpenVMS usage floating_point represents any OpenVMS floating-point data type. Table 11–6 lists all the OpenVMS usages and the PL/I for OpenVMS VAX or PL/I for OpenVMS AXP statements you need to implement them. (The callout numbers in Table 11–6 are keyed to the list following the table.)

Table 11–6 PL/I Implementation of OpenVMS Usages

OpenVMS Data Type	Declaration
access_bit_names	1 ACCESS_BIT_NAMES(32), 2 LENGTH FIXED BINARY(15), 2 DTYPE FIXED BINARY(7) INITIAL((32)DSC$K_DTYPE_T), 2 CLASS FIXED BINARY(7) INITIAL((32)DSC$K_CLASS_S), 2 CHAR_PTR POINTER; **6**

(continued on next page)

Table 11–6 (Cont.) PL/I Implementation of OpenVMS Usages

OpenVMS Data Type	Declaration
	The length of the LENGTH field in each element of the array should correspond to the length of a string of characters pointed to by the CHAR_PTR field. You can use the constants DST$K_CLASS_S and DST$K_DTYPE_T by including the module $DSCDEF from PLI$STARLET or by declaring them GLOBALREF FIXED BINARY(31) VALUE.
access_mode	FIXED BINARY(7) (The constants for this type— PSLC_KERNEL, PSLC_EXEC, PSLC_SUPER, PSLC_USER—are declared in module $PSLDEF in PLI$STARLET.) **1**
address	POINTER
address_range	(2) POINTER **6**
arg_list	1 ARG_LIST BASED, 2 ARGCOUNT FIXED BINARY(31), 2 ARGUMENT (X REFER (ARGCOUNT)) POINTER; **6** If the arguments are passed by value, it may be appropriate to change the type of the ARGUMENT field of the structure. Alternatively, you can use the POSINT, INT, or UNSPEC built-in functions/ pseudovariables to access the data. X should be an expression with a value in the range 0–255 at the time the structure is allocated.
ast_procedure	PROCEDURE or ENTRY **2**
boolean	BIT ALIGNED **1**
byte_signed	FIXED BINARY(7)
byte_unsigned	FIXED BINARY(7) **3**
channel	FIXED BINARY(15)
char_string	CHARACTER(n) **4**
complex_number	(2) FLOAT BINARY(n) or 1 CPLX, 2 REAL FLOAT BIN(n), 2 IMAG FLOAT BIN(n); (See floating_point for values of n.)
cond_value	See module STS$VALUE in PLI$STARLET **6**
context	FIXED BINARY(31)
date_time	BIT(64) ALIGNED **5**
device_name	CHARACTER(n) **4**
ef_cluster_name	CHARACTER(n) **4**
ef_number	FIXED BINARY(31)
exit_handler_block	1 EXIT_HANDLER_BLOCK BASED, 2 FORWARD_LINK POINTER, 2 HANDLER POINTER, 2 ARGCOUNT FIXED BINARY(31), 2 ARGUMENT (n REFER (ARGCOUNT)) POINTER; **6** Replace n with an expression that will yield a value between 0 and 255 at the time the structure is allocated.

(continued on next page)

Table 11–6 (Cont.) PL/I Implementation of OpenVMS Usages

OpenVMS Data Type	Declaration
fab	See module $FABDEF in PLI$STARLET **6**
file_protection	BIT(16) ALIGNED **1**
floating_point	FLOAT BINARY(n) The values for n are as follows: 1 <= n <= 24 - F floating 25 <= n <= 53 - D floating 25 <= n <= 53 - G floating (with /G_FLOAT) 54 <= n <= 113 - H floating
function_code	BIT(32) ALIGNED
identifier	POINTER
io_status_block	Because there are different formats for I/O status blocks for various system services, different definitions will be appropriate for different uses. Some of the common formats are shown here. **6** 1 IOSB_SYS$GETSYI, 2 STATUS FIXED BINARY(31), 2 RESERVED FIXED BINARY(31); 1 IOSB_TTDRIVER_A, 2 STATUS FIXED BINARY(15), 2 BYTE_COUNT FIXED BINARY(15), 2 MBZ FIXED BINARY(31) INITIAL(0); 1 IOSB_TTDRIVER_B, 2 STATUS FIXED BINARY(15), 2 TRANSMIT_SPEED FIXED BINARY(7), 2 RECEIVE_SPEED FIXED BINARY(7), 2 CR_FILL FIXED BINARY(7), 2 LF_FILL FIXED BINARY(7), 2 PARITY_FLAGS FIXED BINARY(7), 2 MBZ FIXED BINARY(7) INITIAL(0);
item_list_2	1 ITEM_LIST_2, 2 ITEM(SIZE), 3 COMPONENT_LENGTH FIXED BINARY(15), 3 ITEM_CODE FIXED BINARY(15), 3 COMPONENT_ADDRESS POINTER, 2 TERMINATOR FIXED BINARY(31) INITIAL(0); **6** Replace SIZE with the number of items you want.

(continued on next page)

Table 11–6 (Cont.) PL/I Implementation of OpenVMS Usages

OpenVMS Data Type	Declaration
item_list_3	1 ITEM_LIST_3, 2 ITEM(SIZE), 3 BUFFER_LENGTH FIXED BINARY(15), 3 ITEM_CODE FIXED BINARY(15), 3 BUFFER_ADDRESS POINTER, 3 RETURN_LENGTH POINTER, 2 TERMINATOR FIXED BINARY(31) INITIAL(0); **6** Replace SIZE with the number of items you want.
item_list_pair	1 ITEM_LIST_PAIR, 2 ITEM(SIZE), 3 ITEM_CODE FIXED BINARY(31), 3 ITEM UNION, 4 INTEGER FIXED BINARY(31), 4 REAL FLOAT BINARY(24), 2 TERMINATOR FIXED BINARY(31) INITIAL(0); **6** Replace SIZE with the number of items you want.
item_quota_list	1 ITEM_QUOTA_LIST, 2 QUOTA(SIZE), 3 NAME FIXED BINARY(7), 3 VALUE FIXED BINARY(31), 2 TERMINATOR FIXED BINARY(7) INITIAL(PQL$_LISTEND); **6** Replace SIZE with the number of quota entries that you want to use. The constant PQL$_LISTEND can be used by including the module $PQLDEF from PLI$STARLET or by declaring it GLOBALREF FIXED BINARY(31) VALUE.
lock_id	FIXED BINARY(31)
lock_status_block	1 LOCK_STATUS_BLOCK, 2 STATUS_CODE FIXED BINARY(15), 2 RESERVED FIXED BINARY(15), 2 LOCK_ID FIXED BINARY(31); **6**
lock_value_block	The declaration of an item of this structure will depend on the use of the structure, because the OpenVMS system does not interpret the value. **6**
logical_name	CHARACTER(n) **4**
longword_signed	FIXED BINARY(31)
longword_unsigned	FIXED BINARY(31) **3**
mask_byte	BIT(8) ALIGNED
mask_longword	BIT(32) ALIGNED
mask_quadword	BIT(64) ALIGNED
mask_word	BIT(16) ALIGNED
null_arg	Omit the corresponding parameter in the call. For example, FOO(A,,B) would omit the second parameter.

(continued on next page)

Table 11–6 (Cont.) PL/I Implementation of OpenVMS Usages

OpenVMS Data Type	Declaration
octaword_signed	BIT(128) ALIGNED **5**
octaword_unsigned	BIT(128) ALIGNED **3** **5**
page_protection	FIXED BINARY(31) (The constants for this type are declared in module $PRTDEF in PLI$STARLET.)
procedure	PROCEDURE or ENTRY **2**
process_id	FIXED BINARY(31)
process_name	CHARACTER(n) **4**
quadword_signed	BIT(64) ALIGNED **5**
quadword_unsigned	BIT(64) ALIGNED **3** **5**
rights_holder	1 RIGHTS_HOLDER, 2 RIGHTS_ID FIXED BINARY(31), 2 ACCESS_RIGHTS BIT(32) ALIGNED; **6**
rights_id	FIXED BINARY(31)
rab	See module $RABDEF in PLI$STARLET **6**
section_id	BIT(64) ALIGNED
section_name	CHARACTER(n) **4**
system_access_id	BIT(64) ALIGNED
time_name	CHARACTER(n) **4**
uic	FIXED BINARY(31)
user_arg	ANY
varying_arg	ANY with OPTIONS(VARIABLE) on the routine declaration
vector_byte_signed	(n) FIXED BINARY(7) **7**
vector_byte_unsigned	(n) FIXED BINARY(7) **3** **7**
vector_longword_signed	(n) FIXED BINARY(31) **7**
vector_longword_ unsigned	(n) FIXED BINARY(31) **3** **7**
vector_quadword_signed	(n) BIT(64) ALIGNED **5** **7**
vector_quadword_ unsigned	(n) BIT(64) ALIGNED **3** **5** **7**
vector_word_signed	(n) FIXED BINARY(15) **7**
vector_word_unsigned	(n) FIXED BINARY(15) **3** **7**
word_signed	FIXED BINARY(15)
word_unsigned	FIXED BINARY(15) **3**

1 System routines are often written so the parameter passed occupies more storage than the object requires. For example, some system services have parameters that return a single bit value in a longword. These variables must be declared BIT(32) ALIGNED (not BIT(1) ALIGNED) so adjacent storage is not overwritten by return values or used incorrectly as input. (Longword Boolean parameters should always be declared BIT(32) ALIGNED.)

2 AST procedures and those passed as parameters of type ENTRY VALUE or ANY VALUE must be external procedures. This applies to all system routines which take procedure parameters, unless explicitly stated otherwise.

3 This is actually an unsigned integer. This declaration is interpreted as a signed number; use the POSINT function to determine the actual value, if necessary.

4 System services require CHARACTER string representation for parameters. Most other system routines allow either CHARACTER or CHARACTER VARYING. For parameter declarations, n should be an asterisk. Note that all system services, RTL routines and utility routines are declared in PLI$STARLET.

5 PL/I for OpenVMS VAX does not support FIXED BINARY numbers with precisions greater than 32. To use larger values, declare variables to be BIT variables of the appropriate size and use the POSINT and SUBSTR bits as necessary to access the values, or declare the item as a structure. The RTL routines LIB$ADDX and LIB$SUBX, which are declared in PLI$STARLET, may be useful if you need to perform arithmetic on these types.

6 Routines declared in PLI$STARLET often use ANY so that you can declare the data structure in the most convenient way for your application. ANY may be necessary in some cases because PL/I does not allow parameter declarations for some data types used by the OpenVMS system.

7 For parameter declarations, the bounds must be constant for arrays passed by reference, or the REFERENCE attribute must be used. For arrays passed by descriptor, asterisks should be used for the array extent instead. (VMS system routines almost always take arrays by reference.)

Note

All system services and many system constants and data structures are declared in PLI$STARLET.TLB.

Also note that while the current version of PL/I for OpenVMS VAX does not support unsigned fixed binary numbers or fixed binary numbers with a precision greater that 31, it is possible that future versions may support these features. If PL/I for OpenVMS VAX is extended to support these types, it is possible that declarations in PLI$STARLET will change to use the new data types where appropriate.

If a system routine argument is optional, it will be indicated in the format section of the routine description in one of two ways:

- [,optional-argument]
- ,[optional-argument]

If the comma appears outside the brackets, you must either pass a zero by value or use a comma in the argument list as a placeholder to indicate the place of the omitted argument. If this is the last argument in the list, you must still include the comma as a placeholder. If the comma appears inside the brackets, you can omit the argument if it is the last argument in the list. Otherwise, you can use a comma in the argument list as a placeholder, and the PL/I for OpenVMS VAX compiler will pass a zero by value for the argument.

To determine the PL/I for OpenVMS VAX parameter descriptors in the declaration of a given system routine, you can display or print the text module for that routine. For example:

```
$ LIBRARY/EXTRACT=SYS$TRNLNM/OUTPUT=LP:TRNLNM -
_$ SYS$LIBRARY:PLI$STARLET/TEXT
```

This LIBRARY command prints the contents of the text module SYS$TRNLNM from the library SYS$LIBRARY:PLI$STARLET. The file is printed on the device LP; the listing file is named TRNLNM.TXT.

11.6.4.3 Symbol Definitions

Many system routines depend on values that are defined in separate symbol definition files. OpenVMS Run-Time Library routines require you to include symbol definitions when you are calling a Screen Management facility routine or a routine that is a jacket to a system service. A *jacket* routine provides a simpler interface to the corresponding system service. For example, the routine LIB$SYS_ASCTIM is a jacket routine for the $ASCTIM system service.

If you are calling a system service, you must include the module $SSDEF to check status. Many system services require other symbol definitions as well. To determine whether you need to include other symbol definitions for the system service you want to use, refer to the documentation for that particular system service. If the documentation states that values are defined in a macro, you must include those symbol definitions in your program.

For example, the description for the *flags* parameter in the SYS$MGBLSC (Map Global Section) system service states that "Symbolic names for the flag bits are defined by the $SECDEF macro." Therefore, when you call SYS$MGBLSC you must include the definitions provided in the $SECDEF module.

In PL/I for OpenVMS VAX a definition module is included as follows:

```
%INCLUDE $SSDEF;
```

You can declare the names of global symbols using the GLOBALREF and VALUE attributes. Then, you can use the names to represent values in an argument list to invoke a system service. For symbolic names that are not defined as OpenVMS global symbols, PL/I for OpenVMS VAX provides text modules in the default INCLUDE library PLI$STARLET.

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.

11.7 Condition Values

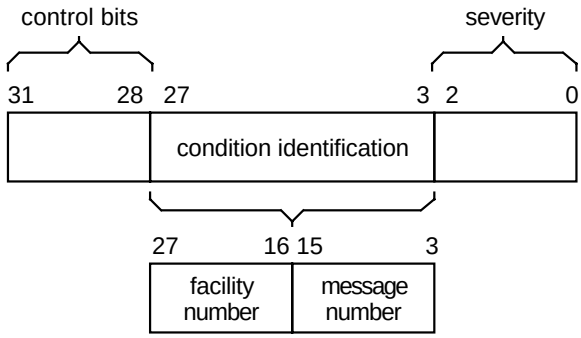
Many system routines return a condition value that indicates success or failure; this value can be either returned or signaled. If a condition value is returned, then you must check the returned value to determine whether the call to the system routine was successful. If a condition value is signaled, then the condition value is signaled to your program instead of being returned as a function value.

Thus, to obtain a condition value from any system routine, you can declare the routine as a function. For example:

```
DECLARE SYS$SETEF ENTRY (FIXED BINARY(31) VALUE)
                  RETURNS (FIXED BINARY(31));
```

This declaration of the SYS$SETEF service allows you to invoke the routine as a function and to obtain a condition value. To provide a unique means of identifying every return condition in the system, fields within the longword value are defined as shown in Figure 11–8.

Figure 11–8 Condition Value Fields



control bits (31–28)
Define special actions to be taken. At present, only bit 28 is used; when set, it inhibits the printing of the message associated with the return status value at image exit. Bits 29 through 31 are reserved for future use by Digital.

facility number (27–16)
Is a unique value assigned to the system component or facility that is returning the status value. Within this field, bit 27 has a special significance. If bit 27 is clear, the facility is a Digital facility: the remaining value in the facility number field is a number assigned by the operating system. If bit 27 is set, the number can indicate a customer-defined facility.

message number (15–3)
Gives an identification number that specifically describes the return status or condition. Within this field, bit 15 has a special significance. If bit 15 is set, then this message number is unique to the facility that is issuing the message. If bit 15 is clear, then this message has been issued by more than one system facility.

severity (2–0)
Specifies a numeric value indicating the severity of the return status. The possible values in these three bits, and their meanings, are as follows:

Value	Meaning
0	Warning
1	Success
2	Error
3	Informational
4	Severe Error
5–7	Reserved

Note that odd values indicate success (an information condition is considered a successful status) and that even values indicate failures (a warning is considered an unsuccessful status).

For testing condition values in a PL/I for OpenVMS VAX program, you can choose to test only whether a procedure completed successfully, or you can test for specific return status values. For either case, you can use the variables declared in the text module $STSDEF. This module is in the default PL/I for OpenVMS VAX text library PLI$STARLET.TLB. 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$MSG,                    /* bits 2 through 15         */
           3 STS$MSG_NO BIT(12),       /* numeric value             */
           3 STS$FAC_SP BIT(1),        /* if set, facility-specific */
         2 STS$FAC,                    /* bits 16 through 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 for OpenVMS VAX program as follows:

```
%INCLUDE $STSDEF;
```

The compiler will automatically locate this module in PLI$STARLET.

To test a condition value for success or failure, you need only test the structure member STS$SUCCESS declared in the structure STS$FIELDS. If this bit is true, then the condition value is a successful value. For example:

```
%INCLUDE SYS$SETPRN;

%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.

11.7.1 Testing for Specific Condition Values

You can also test for specific condition values. Each numeric condition 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 global symbol names for OpenVMS condition values have the following format:

facility$_code

facility
Is an abbreviation or acronym for the system facility that defined the global symbol.

code
Is a mnemonic for the specific condition value.

Some examples of facility codes used in global symbol names follow.

Facility Code	Used by
PLI	PLI for VAX run-time procedures; these status codes are listed in Appendix A.
SS	System services; these status codes are listed in the *OpenVMS System Services Reference Manual*.
DTK	DTK$ Run-Time Library facility.
LIB	LIB$ Run-Time Library facility.
MTH	MTH$ Run-Time Library facility.
OTS	OTS$ Run-Time Library facility.
SMG	SMG$ Run-Time Library facility.
STR	STR$ Run-Time Library facility.
RMS	File system procedures; these status codes are listed in the *OpenVMS Record Management Services Reference Manual*.
SOR	SORT procedures; these status codes are listed in the *VMS Sort/Merge Utility Manual*.

Definitions for the global symbol names for these facilities are located in the default system object module libraries, and thus are automatically located when you link a PL/I for OpenVMS VAX program that references them.

When you write a PL/I for OpenVMS VAX program that calls system procedures and you want to test for specific return status values using the symbol names, you must do the following:

1. Determine, from the documentation of the routine, the return status values that can be returned, and choose the values for which you want to provide specific tests.
2. Include the appropriate module containing the definition from PLI$STARLET or declare the symbolic name for each value of interest as FIXED BINARY(31) and give the variable the GLOBALREF and VALUE attributes. (Note that the first method can be optimized more effectively by the compiler in some cases because the actual values are available at compile time.)

For example, the documentation of the SYS$SETPRN service indicates that it may return the status codes SS$_DUPLNAM (if the name specified as an argument duplicates a name that already exists) and SS$_IVLOGNAM (if the name is invalid; for example, if it contains more than 15 characters). These symbol names can be defined and used as in the following example.

```
%INCLUDE SYS$SETPRN;
%INCLUDE $SSDEF;              /* includes SS$_DUPLNAM and SS$_IVLOGNAM */
%INCLUDE $STSDEF;
   .
   .
   .
STS$VALUE = SYS$SETPRN(NAME);
IF STS$VALUE - SS$_DUPLNAM THEN
   GET LIST(NAME) OPTIONS
                  (PROMPT('Name in use. Reenter: '));
IF STS$VALUE = SS$_IVLOGNAM
   THEN GET LIST(NAME) OPTIONS
                   (PROMPT('Invalid string. Reenter: '));
```

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.

```
%INCLUDE SYS$SETEF;

%INCLUDE $SSDEF;

%INCLUDE $STSDEF;

STS$VALUE = SYS$SETEF(4);

IF ^STS$SUCCESS THEN RETURN (STS$VALUE);1
IF STS$VALUE = SS$_WASSET THEN DO;2
```

In this example, the symbolic name SS$_WASSET is included from PLI$STARLET. The value associated with this condition value is a successful value; it indicates that the flag specified in the routine invocation was previously set.

The routine invocation returns the condition 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 routine returns with the value of STS$VALUE in the RETURN statement. If the service returned with a successful status, the routine continues with an IF statement that checks whether or not the flag was previously set. If so, the DO statement specified in the THEN clause activates the DO-group.

11.7.2 Setting and Displaying Fields Within a Status Value

You can use the structure STS$FIELDS to set or display fields within a status value. For example, to define application-specific message numbers using the format used by VMS, you can specify a facility-wide message number, set the STS$CUST_DEF field to '1'B, assign unique numbers to messages, and define severities for the messages.

Because the fields within this structure are defined as bit strings, and because 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 define a value for STS$VALUE in which the customer-defined facility number is 55 and the unique message number is 14:

```
DECLARE I FIXED BINARY(31);

I = 55;
STS$FAC_NO = UNSPEC(I);
STS$CUST_DEF = '1'B;

I = 14;
STS$MSG_NO = UNSPEC(I);
STS$FAC_SP = '1'B;
```

The intermediate variable I is used to perform the conversions, because the UNSPEC built-in function does not accept constants for arguments.

To set a value for a severity, you must also use the STRING built-in function so that you can set the field to a single value (note that there are two bit-string variables defined within the field SEVERITY). For example:

```
I = 4;
STRING(STS$SEVERITY) = UNSPEC(I);
```

Here, the severity field of STS$VALUE is set to 4.

You can use a similar technique to display fields within a status value. For example, to display the entire facility number (including STS$CUST_DEF), you could write the following:

```
UNSPEC(I) = STRING(STS$FAC);
PUT SKIP LIST('Facility',I);
```

Here, the UNSPEC pseudovariable assigns an integer value to I that represents the bit-string value of the STS$FAC field. You can use the same technique to output the STS$SEVERITY and STS$MSG fields. To display or interpret the STS$FAC_NO or STS$MSG_NO fields, you could use the following:

```
UNSPEC(I) = STS$FAC_NO;
PUT SKIP LIST('Customer facility number',I);
```

You do not need to use the UNSPEC built-in function or the UNSPEC pseudovariable to set or interpret the 1-bit fields STS$SUCCESS, STS$CUST_DEF, STS$FAC_SP, or STS$INHIB_MSG. See the *OpenVMS Messages Utility Reference Manual* for more information on user-defined messages.

11.8 Examples of Calling System Routines

This section provides complete examples of calling system routines, including some SORT/MERGE examples, from PL/I for OpenVMS VAX. In addition to the examples provided here, the *VMS Run-Time Library Routines Volume* and the *OpenVMS System Services Reference Manual* also provide examples for selected routines. Refer to these manuals for help on the use of a specific system routine. For additional information on the OpenVMS SORT/MERGE utility, refer to the *VMS Sort/Merge Utility Manual.*

The system service examples on the next few pages illustrate a number of system service calls. These examples illustrate the following tasks:

- 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 PLI$STARLET to declare the system services. The text of each sample program shows the INCLUDE file for the system service.

All the examples 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 Section 11.7.

11.8.1 Logical Name Translation

Example 11–2 illustrates a call to the Translate Logical Name (SYS$TRNLNM) system service. This system service returns the result of a single logical name translation. In this example, the routine ORION translates the logical name CYGNUS and displays the result on the terminal. If the name is not defined, the routine displays a message indicating that fact. The following notes are keyed to Example 11–2:

1 The procedure declares the logical name to be translated, the logical name table to be searched, a structure that describes to SYS$TRNLNM what information is desired, and where this information should be stored.

2 The reference to SYS$TRNLNM specifies the logical name table, the logical name CYGNUS, and the structure that contains pointers to the variables that will receive the translated logical name length and the logical name. The routine reference does not specify all the arguments. At run time, the argument list for this routine will contain zeros for the omitted arguments.

Example 11–2 Translating a Logical Name

```
ORION: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BINARY(31));

    %INCLUDE SYS$TRNLNM;
    %INCLUDE $LNMDEF;
    %INCLUDE $SSDEF;
    %INCLUDE $STSDEF;

    %REPLACE MAXLEN BY 256;

    DECLARE
       CYGDES CHARACTER(6) STATIC INITIAL ('CYGNUS'),
       NAMETAB CHARACTER(17) STATIC INITIAL ('LNM$PROCESS_TABLE'),
       1 RESULTS,
         2 BUFFER_LENGTH FIXED BINARY(15) INITIAL(MAXLEN),
         2 ITEM_CODE FIXED BINARY(15) INITIAL(LNM$_STRING),
         2 BUFFER_ADDRESS POINTER,
         2 RETURN_LENGTH_ADDRESS POINTER,
         2 TERMINATOR FIXED BINARY(31) INITIAL(0),
       BUFFER CHARACTER(MAXLEN),
       RETURN_LENGTH FIXED BINARY(15);                      1

    RESULTS.BUFFER_ADDRESS = ADDR(BUFFER);
    RESULTS.RETURN_LENGTH_ADDRESS = ADDR(RETURN_LENGTH);
    STS$VALUE = SYS$TRNLNM( , NAMETAB, CYGDES, , RESULTS);  2
    IF STS$VALUE = SS$_NOLOGNAM
    THEN
       PUT SKIP LIST('CYGNUS not defined');
    ELSE
       IF STS$SUCCESS
       THEN
          PUT SKIP LIST('CYGNUS is',SUBSTR(BUFFER,1,RETURN_LENGTH));

    RETURN(STS$VALUE);

    END ORION;
```

11.8.2 Mailbox Services

A mailbox is a virtual I/O device that is used for communication among processes in the system. The routines CREATE_MAILBOX and DELETE_MAILBOX in the following example illustrate the creation and deletion, respectively, of a mailbox.

11.8.2.1 Creating the Mailbox

Example 11–3 illustrates a call to the Create Mailbox and Assign Channel (SYS$CREMBX) system service. This service returns the number of an I/O channel to the calling program. In PL/I for OpenVMS VAX, this number is not needed except for the deletion of the mailbox. You must, however, declare a FIXED BINARY(15) variable to receive the number returned by the system service. The following notes are keyed to Example 11–3:

1 The reference to SYS$CREMBX specifies the permanent flag, output field for the channel number, maximum message size, protection mask, and mailbox logical name. Commas indicate arguments that are not specified; PL/I for OpenVMS VAX places zeros in the argument list in these places.

2 The user privilege to create a permanent mailbox (PRMMBX) is required to call this service with the prmflg argument set to true. A permanent mailbox is not deleted when its creator exits. If this were not a permanent mailbox, the system would automatically delete the mailbox when this procedure is completed.

SYSNAM privilege is required to place a logical name for a mailbox in the system logical name table.

3 The MAXMSG argument to SYS$CREMBX specifies the maximum length of any message sent to the mailbox.

4 The PROMSK argument specifies a 16-bit protection mask for the mailbox. The protection code 'FF00'B4 restricts access to the owner and the system and denies access to the group and the world. Note that the value specified as 'FF00' to PL/I for OpenVMS VAX actually sets the low 8 bits of the parameter given the internal representation of bit-strings in PL/I for OpenVMS VAX.

Example 11–3 Creating a Mailbox

```
CREATE_MAILBOX: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BINARY(31));

    %INCLUDE SYS$CREMBX;
    %INCLUDE $STSDEF;
    %REPLACE MESSAGE_SIZE BY 132;
    %REPLACE PERMANENT BY '1'B;

    DECLARE
        CHANNEL FIXED BINARY(15),
        MAILBOX_NAME CHARACTER(11)
                    STATIC INITIAL('PLI_MAILBOX');
    /*
     * Call SYS$CREMBX omitting optional arguments.
     * (Note that trailing optional arguments cannot
     * be omitted for system services unless specifically
     * indicated in the service description.)
     */
    STS$VALUE = SYS$CREMBX(                              1
                        PERMANENT,                       2
                        CHANNEL,
                        MESSAGE_SIZE,                    3
                        'FF00'B4,,                       4
                        MAILBOX_NAME);
    /*
     * Return to command level with status. If SYS$CREMBX
     * completed with an error, the appropriate message is
     * displayed at the command level.
     */
    RETURN(STS$VALUE);
    END CREATE_MAILBOX;
```

11.8.2.2 Deleting the Mailbox

Example 11–4 illustrates a call to the Delete Mailbox (SYS$DELMBX) system service. The procedure DELETE_MAILBOX deletes the mailbox PLI_MAILBOX. A mailbox is deleted when a channel number is specified; this program assigns a channel to the mailbox to obtain a number to be specified in deleting the mailbox. The following notes are keyed to Example 11–4:

1 The procedure declares the system services Assign I/O Channel (SYS$ASSIGN) and Delete Mailbox (SYS$DELMBX). The channel number output by SYS$ASSIGN is used as input to SYS$DELMBX.

2 The call to SYS$ASSIGN specifies the logical name of the mailbox and the variable CHANNEL to allow the system service to return a channel number. The optional arguments that are not specified in this call are represented by the required commas at the end of the argument list.

3 The channel number output by SYS$ASSIGN is passed to SYS$DELMBX by value. The mailbox PLI_MAILBOX is deleted when all programs that have opened the mailbox have closed it.

Example 11–4 Deleting a Mailbox

(continued on next page)

Example 11–4 (Cont.) Deleting a Mailbox

```
DELETE_MAILBOX: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BINARY(31));

    %INCLUDE SYS$ASSIGN;                                    1
    %INCLUDE SYS$DELMBX;
    %INCLUDE $STSDEF;

    DECLARE
        MAILBOX_NAME CHARACTER(11)
                STATIC READONLY INITIAL('PLI_MAILBOX')
        CHANNEL FIXED BINARY(15);
    /*
     * Call SYS$ASSIGN and check return; if not successful exit
     */
    STS$VALUE = SYS$ASSIGN(MAILBOX_NAME,CHANNEL,,,);    2
    IF ^STS$SUCCESS
    THEN
       RETURN(STS$VALUE);
    /*
     * Call SYS$DELMBX and check return
     */
    STS$VALUE = SYS$DELMBX(CHANNEL);                        3
    RETURN(STS$VALUE);

    END DELETE_MAILBOX;
```

11.8.3 Timer and Time Conversion Routines

The system services that depend on time, either an absolute time or a delta time, refer to a time value that is maintained in a 64-bit field. There are system services that convert a character string that specifies a time to its binary equivalent and vice versa.

11.8.3.1 Obtaining a Time Value in System Format

The PL/I for OpenVMS VAX procedure GETBINTIM, shown in Example 11–5, accepts a character-string time value as a parameter and returns the binary time value to the point of the procedure's invocation. The following notes are keyed to Example 11–5:

1 GETBINTIM declares the system service SYS$BINTIM, which converts an ASCII string to a binary time value.

2 GETBINTIM invokes SYS$BINTIM as a function and tests the return status. An error results if the ASCII time value is not specified correctly. When an error is returned, GETBINTIM returns a zero to the point of the procedure's invocation.

This procedure may be invoked as follows to supply a date and time value for a file in an ENVIRONMENT option:

```
DECLARE GETBINTIM ENTRY( CHAR(*) ) RETURNS BIT(64) ALIGNED,
        (CREATED_DATE,EXPIRE_DATE) BIT(64) ALIGNED;

    CREATED_DATE = GETBINTIM('17-JUN-1985 00:00:00.00');
    EXPIRE_DATE = GETBINTIM('31-DEC-1991 00:00:00.00');
    OPEN FILE(TAPEFILE) ENVIRONMENT(

           CREATION_DATE(CREATED_DATE),
           EXPIRATION_DATE(EXPIRE_DATE));
```

Example 11–5 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;                                  1
    %INCLUDE $STSDEF;

    DECLARE
        ASCII_STRING CHARACTER(*),
        BINARY_TIME BIT(64) ALIGNED;
    /*
     * If successful, return binary time to point of
     * invocation. Otherwise, return 0 -- this results
     * in absolute time 17-NOV-1858.
     */
    STS$VALUE = SYS$BINTIM(ASCII_STRING,BINARY_TIME);  2
    IF STS$SUCCESS
    THEN
        RETURN(BINARY_TIME);
    ELSE
        RETURN((64)'0'B);
    END GETBINTIM;
```

11.8.3.2 Setting the Timer

The procedure in Example 11–6 uses the Set Timer (SYS$SETIMR) system service. It issues a time request for some activity to occur in 10 seconds and specifies the number of an event flag to be set when the 10 seconds have elapsed. The following notes are keyed to Example 11–6:

1 This procedure uses the GETBINTIM function to convert an ASCII time value to the system's 64-bit format.

2 LIB$GET_EF allocates an event flag from a process-wide pool and returns the event flag number.

3 The procedure invokes SYS$SETIMR, specifying by its first argument that SYS$SETIMR should set the specified event flag when the time expires. The argument list contains a reference to GETBINTIM, which returns the system time value for 10 seconds.

4 The procedure uses the Wait for Event Flag (SYS$WAITFR) system service to wait for the event flag specified in the call to SYS$SETIMR. When the flag is set, the procedure displays a message and exits.

Example 11–6 Setting a Timer

(continued on next page)

Example 11–6 (Cont.) Setting a Timer

```
SET_TIMER: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BINARY(31));
    %INCLUDE LIB$GET_EF;
    %INCLUDE LIB$FREE_EF;
    %INCLUDE SYS$SETIMR;
    %INCLUDE SYS$WAITFR;
    %INCLUDE $STSDEF;
    DECLARE
        GETBINTIM ENTRY(CHAR(*))  /* character string time */   1
            RETURNS(BIT(64) ALIGNED);
    DECLARE EVENT_FLAG_NUM FIXED BIN(31);
    /*
     * Get an event flag to use.
     */
    STS$VALUE = LIB$GET_EF(EVENT_FLAG_NUM);                    2
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);
    /*
     * Set the timer for 10 seconds.
     */
    STS$VALUE = SYS$SETIMR(EVENT_FLAG_NUM,
                    GETBINTIM('0 00:00:10'),,);                3
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);
    /*
     * Wait for the event flag, and display a
     * message when the timer completes.
     */
    STS$VALUE = SYS$WAITFR(EVENT_FLAG_NUM);                    4
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);
    PUT SKIP LIST('Timer up!');
    /*
     * Release the event flag.
     */
    STS$VALUE = LIB$GET_EF(EVENT_FLAG_NUM);
    RETURN(STS$VALUE);
    END SET_TIMER;
```

11.8.4 A Ctrl/c-Handling Routine

A Ctrl/c routine is a subroutine that is given control when the execution of the program is interrupted externally by the Ctrl/c function. To enable a Ctrl /c routine, you must code a call to the SYS$QIO (Queue I/O Request) system service, which performs I/O. In this call to SYS$QIO, you specify the name of an external procedure that will be executed when the interruption occurs. This type of procedure is called an asynchronous system trap (AST) routine because it may be executed at any time.

The sample programs in this section interact as follows:

- The procedure SET_CTRLC in Example 11–7 establishes the Ctrl/c routine. It calls SYS$ASSIGN and SYS$QIO and specifies the name of the external AST routine, C_AST.
- The procedure C_AST in Example 11–8 is the AST routine itself. C_AST sets the CNTRL_C_INTER bit to signal that an interrupt has occurred.

- The test program TESTC in Example 11–9 calls SET_CTRLC to establish the Ctrl/c handler. Because it is a test program, it does not do any more than enable the Ctrl/c handler, place itself in an infinite loop, and signal when a Ctrl/c interrupt occurs. The execution of this program must be interrupted by Ctrl/y. If you create your own versions of this program, you should note the different effects of Ctrl/c and Ctrl/y.

11.8.4.1 Establishing a Ctrl/c-Handling Routine

The following notes are keyed to Example 11–7:

1 SET_CTRLC includes the declarations for the SYS$ASSIGN, SYS$QIO, and SYS$WAITFR system services. The SYS$QIO system service requires as an argument a channel number, that is, an I/O path to a device. The SYS$ASSIGN system service obtains a channel number.

2 The I/O function codes IO$_SETMODE and IO$M_CTRLCAST are declared in $IODEF, which is obtained from PLI$STARLET.TLB. In a call to SYS$QIO, the specific I/O request is indicated by a symbolic name. These names have the following meanings:

- IO$_SETMODE is a function code that specifies a type of I/O request; it sets the terminal mode.
- IO$M_CTRLCAST is a function modifier that indicates the mode setting performed by this request, that is, to enable Ctrl/c interrupts.

These function codes and modifiers must be ORed together to obtain the correct result, as shown in the invocation of SYS$QIO (Note 9).

3 The variable TTCHAN receives the channel number assigned to the current terminal device.

4 An I/O status block is an 8-byte structure that is filled with status information when an I/O request is completed. The first two bytes always contain the status of the I/O request.

5 To define a Ctrl/c AST routine, the name of the entry is passed as an argument to SYS$QIO. In PL/I for OpenVMS VAX, this must be the name of an external entry, because it must be passed by immediate value.

When an AST routine is specified in a system service, you have the option of specifying an argument to be passed as a parameter of the AST routine. AST parameters are always passed by value in the argument list. For an AST routine written in PL/I for OpenVMS VAX to correctly interpret the parameter, the AST routine must receive the parameter by reference—this means that the AST parameter must be passed by a pointer to its value.

6 The CNTRL_C_INTER bit is declared and set to zero. This flag is used to call a procedure that signals that an interrupt has occurred.

7 The variable IO_SUCCESS is used to ensure that the I/O completed successfully.

8 The procedure calls SYS$ASSIGN to assign a channel to the current terminal. The simplest way to do this when the terminal may not always be the same physical device is to specify the logical name TT. When the service completes successfully, the variable TTCHAN contains the terminal channel number.

9 The call to SYS$QIO specifies an event flag, the channel number, the I/O function to be performed for the device, the address of the I/O status block, and the name of the AST entry.

SYS$QIO does not actually perform an I/O operation, but merely queues it, as its name suggests. A successful return from SYS$QIO indicates that the request is queued. Proper programming practice requires that the caller of SYS$QIO either wait until the I/O completes, or request notification of completion by the execution of an AST routine. In this example, the procedure waits for the event flag specified in the call to SYS$QIO. When the event flag is set, the I/O is completed. Note that it is generally advisable to use the RTL routines LIB$GET_EF and LIB$FREE_EF to avoid the overlap of event flags. Several other examples in this section show the use of these routines.

10 Following the call to SYS$QIO, the procedure waits until the request is actually performed and then tests the status value in the I/O status block.

Example 11–7 Establishing a Ctrl/c Routine

```
SET_CTRLC: PROCEDURE RETURNS(FIXED BINARY(31));
    %INCLUDE SYS$ASSIGN;                                          1
    %INCLUDE SYS$QIO;
    %INCLUDE SYS$WAITFR;
    %INCLUDE $IODEF;                                              2
    %INCLUDE $STSDEF;

    DECLARE TTCHAN FIXED BINARY(15);                              3

    DECLARE
        1 IOSB,                                                   4
          2 VALUE FIXED (15),         /* Return status */
          2 NOT_USED(3) FIXED (15),
        C_AST ENTRY(POINTER);    /* CTRL/C AST routine */         5

    DECLARE
        CNTRL_C_INTER STATIC BIT(1) ALIGNED GLOBALDEF,            6
        IO_SUCCESS BIT(1) ALIGNED BASED(ADDR(IOSB.VALUE));        7

    DECLARE VALUE BUILTIN;

    /*
     * 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,,);                       8
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);

    STS$VALUE = SYS$QIO (1,TTCHAN,
                IO$_SETMODE|IO$M_CTRLCAST, /* function */  9
                IOSB,                   /* I/O status block */
                ,,                   /* omit QIO AST argument */
                VALUE(C_AST),  /* AST routine for IO$_CTRLCAST */
                ,,,,);           /* unspecified p2 through p6 */
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);

    STS$VALUE = SYS$WAITFR(1);
    IF ^IO_SUCCESS
    THEN RETURN(IOSB.VALUE);                                      10

    CNTRL_C_INTER = '0'B;
    RETURN(1);
    END SET_CTRLC;
```

11.8.4.2 Ctrl/c Routine

The following description refers to Example 11–8.

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. The C_AST interrupt routine sets the CNTRL_C_INTER bit. When control is returned to the main routine, a Ctrl/c interrupt message is printed out. In addition, the Ctrl/c handler is reenabled by calling SET_CTRLC.

Example 11–8 Ctrl/c Handler

```
C_AST: PROCEDURE;
    DECLARE CNTRL_C_INTER STATIC BIT(1) ALIGNED GLOBALREF;
    CNTRL_C_INTER = '1'B;
    END C_AST;
```

11.8.4.3 Testing the Ctrl/c Routine

The procedure TESTC, in Example 11–9, tests the SET_CTRLC and C_AST routines. The techniques used here can be applied to any procedure in which you want to detect and respond to an external interrupt via Ctrl/c.

Example 11–9 Testing the Ctrl/c Routine

```
TESTC: PROCEDURE OPTIONS(MAIN) RETURNS(FIXED BIN(31));

    /*
     * Field declarations for Return Status Values.
     */
    %INCLUDE $STSDEF;

    DECLARE SET_CTRLC ENTRY RETURNS(FIXED BIN(31));          1
    DECLARE CNTRL_C_INTER BIT(1) GLOBALREF;
    %REPLACE TRUE BY '1'B;

    SIGNAL_INTER: PROCEDURE;                                 2
        PUT SKIP LIST('Control/C interrupt');
        STS$VALUE = SET_CTRLC();   /* reenable CTRL/C handler */
        END;
    STS$VALUE = SET_CTRLC_();                                3
    IF ^STS$SUCCESS
    THEN
        RETURN(STS$VALUE);
    DO WHILE (TRUE);                                         4
        IF CNTRL_C_INTER
        THEN
            CALL SIGNAL_INTER;
        END;

    END TESTC;
```

The following notes are keyed to Example 11–9:

1 The procedure declares the external routine SET_CTRLC and the CNTRL_C_INTER variable.

2 If SIGNAL_INTER is called, a message is printed and SET_CTRLC is called to reenable the Ctrl/c handler.

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.

3 The procedure calls SET_CTRLC to establish the Ctrl/c handler.

4 The procedure places itself in an infinite loop. Each time Ctrl/c is entered, the procedure displays its message for the Ctrl/c interrupt and continues.

Note that when this program is run, it can be interrupted at the terminal and stopped only by the Ctrl/y function.

11.8.5 Obtaining Job/Process Information

The Get Job/Process Information (SYS$GETJPI) system service returns information about a specific aspect or attribute of a job that is currently being executed. A call to this service requires that you set up two buffers:

- A buffer called an item list, which specifies what items of information you want
- An output buffer to receive each item of information

The procedure TIME, shown in Example 11–10, uses SYS$GETJPI to acquire performance statistics about the execution of a program. It has two entry points:

- The entry TIMRB is invoked at the beginning of the time for which statistics are to be accumulated.
- The entry TIMRE is invoked at the end of the time for which statistics are to be accumulated.

The statistics that are displayed represent the differences between the values acquired at the entry TIMRE and those acquired at the entry TIMRB.

The following notes are keyed to Example 11–10:

1 The module $JPIDEF contains the definitions of the constant identifiers whose names and values correspond to the item codes required for SYS$GETJPI.

2 The structure JPI_LIST contains a minor structure for each item requested. Each minor structure has the same required format; it contains the following information:

- The length, in bytes, of the buffer you have declared for receiving the information
- A numeric code, specified symbolically, that indicates the information requested
- A pointer to the buffer you have declared for receiving the information
- A variable which, if nonzero, must contain a pointer to a variable that will receive the length of the information returned by SYS$GETJPI

The list is terminated by a longword containing zero.

3 The item codes for SYS$GETJPI are specified using the constant identifiers in the INCLUDE file for $JPIDEF. Each constant identifier specifies a unique numeric code that SYS$GETJPI uses to determine the information to be returned.

4 Variables are declared for the return information for TIMRB and TIMRE. The FORTRAN procedure FOR$SECNDS is in the system run-time procedure library. It returns the current system time in seconds.

5 At TIMRE and TIMRB, the item list for each item is initialized with a pointer to the appropriate return field.

6 SYS$GETJPI is invoked as a function whose value is compared to the status code SS$_NORMAL. If they do not match, the procedure exits with a message.

7 When entered at TIMRE, the procedure obtains the current information, calculates the differences in the statistics required by TIMRB and those obtained in the most recent call to SYS$GETJPI, and displays the results on the terminal. Then, it falls through to TIMRB to ensure that the fields are reinitialized.

8 When entered at TIMRB, the start values for the statistics are initialized with the current values obtained from SYS$GETJPI, and the procedure exits.

Example 11–10 TIMRE and TIMRB

```
TIME: PROCEDURE;

    %INCLUDE SYS$GETJPI;

    /*
     * INCLUDE definitions required by SYS$GETJPI
     */
    %INCLUDE $JPIDEF;     /* item codes */                    1
    %INCLUDE $SSDEF;   /* System (SS$_*) status values */
    %INCLUDE $STSDEF;   /* status value variable */

    DECLARE
        1 JPI_LIST STATIC EXTERNAL,                           2
          2 JPI_BUFIO,        /* Buffered I/O count */
            3 LENGTH FIXED BIN(15) INITIAL(4),
            3 ITMCOD FIXED BIN(15) INITIAL(JPI$_BUFIO),       3
            3 BUFADR POINTER INITIAL(NULL()),
            3 RETLEN POINTER INITIAL(NULL()),
          2 JPI_CPUTIM,          /* CPU time */
            3 LENGTH FIXED BIN(15) INITIAL(4),
            3 ITMCOD FIXED BIN(15) INITIAL(JPI$_CPUTIM),      3
            3 BUFADR POINTER INITIAL(NULL()),
            3 RETLEN POINTER INITIAL(NULL()),
          2 JPI_DIRIO          /* Direct I/O count */
            3 LENGTH FIXED BIN(15) INITIAL(4),
            3 ITMCOD FIXED BIN(15) INITIAL(JPI$_DIRIO),       3
            3 BUFADR POINTER INITIAL(NULL()),
            3 RETLEN POINTER INITIAL(NULL()),
          2 JPI_PAGEFLTS          /* Page faults */
            3 LENGTH FIXED BIN(15) INITIAL(4),
            3 ITMCOD FIXED BIN(15) INITIAL(JPI$_PAGEFLTS),    3
            3 BUFADR POINTER INITIAL(NULL()),
            3 RETLEN POINTER INITIAL(NULL()),
          2 ENDLIST FIXED BIN(31) INITIAL(0);

    DECLARE
        (TO,CLOCK_TIME) FLOAT BIN(24) STATIC EXTERNAL,        4
        (BUFIO,END_BUFIO,CPUTIM,END_CPUTIM,DIRIO,
            END_DIRIO,PAGEFLTS,END_PAGEFLTS)
                    FIXED BIN(31) STATIC EXTERNAL,
        CPUSECONDS FLOAT BIN(24);

DECLARE FOR$SECNDS ENTRY (FLOAT BIN(24)) RETURNS(FLOAT BIN(24));
```

(continued on next page)

Example 11–10 (Cont.) TIMRE and TIMRB

```
TIMRE:        ENTRY;
    JPI_BUFIO.BUFADR = ADDR(END_BUFIO);                          5
    JPI_CPUTIM.BUFADR = ADDR(END_CPUTIM);
    JPI_DIRIO.BUFADR = ADDR(END_DIRIO);
    JPI_PAGEFLTS.BUFADR = ADDR(END_PAGEFLTS);

    IF SYS$GETJPIW(,,,JPI_LIST,,,)^=SS$_NORMAL                   6
    THEN
        PUT SKIP LIST ('Error from SYS$GETJPI');

    CLOCK_TIME = FOR$SECNDS(TO);
    CPUSECONDS = (END_CPUTIM-CPUTIM)/100E0;
    BUFIO = END_BUFIO-BUFIO;                                     7
    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 reinitialize.
     */
TIMRB:        ENTRY;                                             8
    TO = FOR$SECNDS(0E0);
    JPI_BUFIO.BUFADR = ADDR(BUFIO);                              5
    JPI_CPUTIM.BUFADR = ADDR(CPUTIM);
    JPI_DIRIO.BUFADR = ADDR(DIRIO);
    JPI_PAGEFLTS.BUFADR = ADDR(PAGEFLTS);

    IF SYS$GETJPIW(,,,JPI_LIST,,,)^=SS$_NORMAL
    THEN
        PUT SKIP LIST ('Error from SYS$GETJPI');

    RETURN;

    END TIME;
```

11.8.6 Using SORT Routines

Example 11–11 shows a sample procedure that calls the SORT routines to perform an alphabetic sort on a file. The following notes are keyed to Example 11–11:

1 The definitions for the SORT routines and several groups of constants are included from PLI$STARLET.

2 The input and output file specifications are character-string arguments, initialized to the logical names INFILE and OUTFILE. INFILE must be equated to a file whose records are no longer than 80 characters.

3 The key buffer specifies the information required for SOR$BEGIN_SORT.

4 The SORT routines that are required to sort a file must be invoked in this order:

a. SOR$PASS_FILES specifies the input and output file specifications. These can be logical names.

b. SOR$BEGIN_SORT specifies the sizes of the records, key data types, sort sequences, and so forth.

c. SOR$SORT_MERGE initiates the sorting.

d. SOR$END_SORT calls SORT to clean up its work areas and close its temporary files.

5 The procedures are invoked in the order listed in note 1. Each procedure returns its return value to STS$VALUE. If there are errors, the procedure returns with the value of STS$VALUE.

Example 11–11 Sorting Files

```
/*
 * Sort a file
 */
SORTEM: PROCEDURE RETURNS(FIXED BINARY(31));

    /*
     * Include the declarations of the SORT procedures required
     * for a sort using the file interface.
     */
    %INCLUDE SOR$PASS_FILES; /* SORT File Spec Procedure */    1
    %INCLUDE SOR$BEGIN_SORT;    /* SORT Init Procedure */
    %INCLUDE SOR$SORT_MERGE; /* Procedure to Initiate Sort */
    %INCLUDE SOR$END_SORT;   /* Sort Termination Procedure */

    /*
     * Include constants and return status variable.
     */
    %INCLUDE $DSCDEF;    /* Include data type definitions */
    %INCLUDE $FABDEF;         /* FAB declarations */
    %INCLUDE $STSDEF;  /* Declarations for return status value */

    /*
     * Additional constants not currently available in PLI$STARLET.
     * (SORT constants described in the SOR$BEGIN_SORT documentation.)
     */
    %REPLACE ASCENDING_ORDER BY 0;
    %REPLACE DESCENDING_ORDER BY 1;

    /*
     * 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'),       2
        OUTPUT_FILE CHARACTER(7) STATIC INIT('OUTFILE');
```

(continued on next page)

Example 11–11 (Cont.) Sorting Files

```
    /*
     * Declare the key buffer array required to sort the first 80
     * characters of any record. (Note that while the SORT documentation
     * describes this as an array, it is more obviously expressed in
     * PL/I as a structure. An array of FIXED BIN(15) elements could
     * be used instead.)
     */
    DECLARE
        1 KEY_BUFFER STATIC,
          2 NUMBER_OF_KEYS FIXED BINARY(15) INIT(1),           3
          2 KEY_TYPE FIXED BINARY(15) INIT(DSC$K_DTYPE_T), /* character */
          2 KEY_ORDER FIXED BINARY(15) INIT(ASCENDING_ORDER),
          2 START_POS FIXED BINARY(15) INIT(0),
          2 KEY_LENGTH FIXED BINARY(15) INIT(80),
        LONGEST_RECORD FIXED BINARY(15) STATIC INIT(80);

    /*
     * Call the SORT routines in the required order.           4
     * After each call to SORT, check STS$SUCCESS.
     */
    STS$VALUE = SOR$PASS_FILES(
                        INPUT_FILE,    /* Input file name */
                        OUTPUT_FILE,   /* Output file name */
                        FAB$C_REL,     /* File organization */
                        FAB$C_VAR);    /* Record type */
    IF ^STS$SUCCESS
    THEN
        GOTO ERROR;

    STS$VALUE = SOR$BEGIN_SORT(KEY_BUFFER,LONGEST_RECORD);
    IF ^STS$SUCCESS
    THEN
        GOTO ERROR;

    STS$VALUE = SOR$SORT_MERGE();
    IF ^STS$SUCCESS
    THEN
        GOTO ERROR;

    STS$VALUE = SOR$END_SORT();
    IF ^STS$SUCCESS
    THEN
        GOTO ERROR;
    RETURN(1);

ERROR:                                                      5
    PUT SKIP(2) EDIT ('SORT Failed. Error Code',STS$VALUE) (A,X,F(8));
    RETURN(STS$VALUE);

    END SORTEM;
```

Example 11–12 shows a procedure that performs a record sort, processing each record before passing it to the SORT program. The following notes are keyed to Example 11–12:

1 A record sort requires that SORT routines be called in the following order:

a. SOR$BEGIN_SORT specifies the key data types, record sizes, collating sequence, and so on, in a key buffer area.

b. SOR$RELEASE_REC passes each record to SORT.

c. SOR$SORT_MERGE requests SORT to perform the sort on the records it receives.

d. SOR$RETURN_REC requests SORT to pass a single record back. SORT returns the records in sorted order.

e. SOR$END_SORT finishes the SORT.

2 $SSDEF contains the symbol SS$_ENDOFFILE. SORT returns this value when SOR$RETURN_REC requests a record after all records have been returned.

3 Within the key buffer, the START_POS field indicates that the key field within each record begins in position 25, that is, the capital field (note that this is equivalent to an offset of 24 bytes).

4 The LONGEST_RECORD variable specifies the value SIZE(STATE_RECORD). This is the length of each record in the file, plus the length of the key on which the records are to be sorted.

5 The structure STATE_RECORD contains the key field on which the records are to be sorted, as well as the structure declaration of the records in the file STATE_FILE.

6 STATE_RECORD_CHAR is a character string that overlays the STATE_RECORD structure. It is used to pass records to SOR$RELEASE_REC and to obtain records from SOR$RETURN_REC.

7 The procedure declares input and output files, and calls SOR$BEGIN_SORT to begin the sorting process.

8 The records are passed to SOR$RELEASE_REC.

9 SOR$SORT_MERGE is invoked to perform the merge.

10 SOR$RETURN_REC returns each record individually, without the key field, to the structure STATE. When there are no more records, SOR$RETURN_REC returns with the function value SS$_ENDOFFILE.

11 The sort is completed.

Example 11–12 A Record Sort

```
/*
 * This progam 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 routine
     */
    %INCLUDE $DSCDEF;          /* Include data type definitions */
    %INCLUDE SOR$BEGIN_SORT;   /* SORT init procedure */       1
    %INCLUDE SOR$RELEASE_REC;  /* SORT procedure to send records */
    %INCLUDE SOR$SORT_MERGE;   /* Procedure to initiate SORT */
    %INCLUDE SOR$RETURN_REC;   /* SORT procedure to retrieve records */
    %INCLUDE SOR$END_SORT;     /* SORT termination procedure */
    %INCLUDE $STSDEF;          /* Declare return status values */
    %INCLUDE $SSDEF;           /* Status codes */              2

    DECLARE EOF BIT(1) INIT('0'B);
```

(continued on next page)

Example 11–12 (Cont.) A Record Sort

```
/*
 * 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(DSC$K_DTYPE_T), /* char keys */
        2 KEY_ORDER FIXED BINARY(15) INIT(0),   /* ascending order */
        2 START_POS FIXED BINARY(15) INIT(24),              3
        2 KEY_LENGTH FIXED BINARY(15) INIT(20),
     LONGEST_RECORD FIXED BINARY(15)                        4
                 INIT(SIZE(STATE_RECORD));

/*
 * Declare a buffer to construct each record to be passed to SORT
 */
DECLARE 1 STATE_RECORD STATIC,  /* complete record */       5
          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),
    STATE_RECORD_CHAR CHARACTER(SIZE(STATE_RECORD))         6
                BASED(ADDR(STATE_RECORD));

/*
 * Input and output files
 */
DECLARE STATE_FILE FILE INPUT RECORD SEQUENTIAL,
        SORTED_FILE FILE RECORD OUTPUT SEQUENTIAL;          7

/*
 * Call SOR$BEGIN_SORT
 */
STS$VALUE = SOR$BEGIN_SORT(KEY_BUFFER,LONGEST_RECORD);
IF ^STS$SUCCESS
THEN RETURN(STS$VALUE);

/*
 * Enter DO-loop to read the input file STATE_FILE.
 * Then call SOR$RELEASE_REC.
 */
OPEN FILE(STATE_FILE);
ON ENDFILE(STATE_FILE) EOF = '1'B;
READ FILE(STATE_FILE) INTO(STATE_RECORD);
DO WHILE (^EOF);
    STS$VALUE = SOR$RELEASE_REC(                            8
                        STATE_RECORD_CHAR);
    IF ^STS$SUCCESS
    THEN RETURN(STS$VALUE);
    READ FILE(STATE_FILE) INTO(STATE_RECORD);
    END;
CLOSE FILE(STATE_FILE);
PUT SKIP LIST('**** ALL RECORDS RELEASED');
```

(continued on next page)

Example 11–12 (Cont.) A Record Sort

```
/*
 * Call SOR$SORT_MERGE to sort the records that were released
 */
STS$VALUE = SOR$SORT_MERGE();                                   9
IF ^STS$SUCCESS
THEN RETURN(STS$VALUE);

/*
 * Loop through the DO-group to get back each record and
 * write it to the sorted output file.
 */
STS$VALUE = 1;
OPEN FILE(SORTED_FILE) OUTPUT;
DO WHILE (STS$VALUE ^=SS$_ENDOFFILE);
    STS$VALUE = SOR$RETURN_REC(STATE_RECORD_CHAR);             10
    IF STS$SUCCESS
    THEN WRITE FILE(SORTED_FILE) FROM(STATE_RECORD);
    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$END_SORT();                                     11
IF ^STS$SUCCESS
THEN RETURN(STS$VALUE);
RETURN(1);                  /* successful completion */
END;
```

12
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. PL/I for OpenVMS VAX and PL/I for OpenVMS AXP provide an alternative method for defining external variables. Using the GLOBALDEF attribute, one module can 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 discusses the following topics:

- Using global symbols within PL/I procedures
- The READONLY and VALUE attributes
- Declaring and using system-defined global symbols

12.1 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 12–1 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 15.

Table 12–1 Comparison of Global Symbols and External Variables

Global Symbols	External Variables
Declared with the GLOBALDEF and GLOBALREF attributes.	Declared with the EXTERNAL attribute.
Can be initialized only in the module that defines it with the GLOBALDEF attribute. All other modules must specify GLOBALREF.	Must be declared with the EXTERNAL attribute in all modules that declare it. If initialized, must be initialized with the same value in all modules that declare it.

(continued on next page)

Table 12–1 (Cont.) Comparison of Global Symbols and External Variables

Global Symbols	External Variables
Correspond to global symbols declared in assembly language.	Correspond to FORTRAN common blocks.
Fixed-point binary or bit string (less than 33 bits) global symbols can have the VALUE attribute.	Cannot have the VALUE attribute.
No practical limit on the number of global symbols that can be defined and referenced in an object module.	Limited to 254 external variables in an object module (minus the number of external file constants).
Allocation of storage can be controlled by explicit specification of a program section name.	No control over storage allocation. Each variable is placed in a separate program section.

12.1.1 The GLOBALDEF Attribute

The GLOBALDEF attribute declares an external variable or an external file constant. You can optionally control the program section in which the data is allocated.

The format of the GLOBALDEF attribute is as follows:

```
GLOBALDEF [ (psect-name) ]
```

psect-name
Specifies the name of a program section. A program section name can contain up to 31 alphanumeric characters, including dollar signs ($) and underscores (_). The first character cannot be a numeric character (0 through 9).

If you do not specify a program section name, PL/I places the definition for the name in the default program section associated with the variable. For information on program sections created by PL/I, see Chapter 15.

The GLOBALDEF attribute implies the EXTERNAL and STATIC attributes.

The following restrictions apply to the use of the GLOBALDEF attribute:

- The GLOBALDEF attribute conflicts with the GLOBALREF and INTERNAL attributes.
- It cannot be used with ENTRY constants.
- Only one procedure in a program can declare a particular external variable with the GLOBALDEF attribute.

12.1.2 The GLOBALREF 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.

12.1.3 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 the following lines:

```
ABC:  PROCEDURE;
   DECLARE UNIQUE_VALUE GLOBALDEF FIXED BINARY
       INITIAL (60);
   DECLARE XYZ EXTERNAL ENTRY (CHARACTER (*));
   .
   .
   .
   CALL XYZ ('STRING');
```

The procedure XYZ contains the following lines:

```
XYZ:  PROCEDURE (STRING_VAL);
   DECLARE UNIQUE_VALUE GLOBALREF FIXED BINARY;
   .
   .
   .
```

In these examples, 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.

12.1.4 Using MACRO Global Symbols with Multiple Definitions

Using the VAX MACRO programming language, it is possible to give a global external variable more than one name. However, in a PL/I procedure, you can use only one global symbol name for a particular variable. PL/I assumes that distinct global symbol names denote distinct storage locations; the storage associated with different names must not overlap. This rule applies only to global symbols that are declared without the VALUE attribute.

12.2 The READONLY and VALUE Attributes

PL/I for OpenVMS VAX and PL/I for OpenVMS AXP define two storage class attributes that are not in the standard PL/I language: READONLY and VALUE. The READONLY attribute can be specified for any static variable. The VALUE attribute can be specified only for variables that are declared with the GLOBALREF or GLOBALDEF attributes. You cannot declare a variable with both the READONLY and VALUE attributes.

12.2.1 The READONLY Attribute

The READONLY attribute can be applied to any static variable whose value will not change during the program execution. For example, you can initialize fixed values with the PL/I attributes STATIC and INITIAL, and use the READONLY attribute as in this example:

```
DECLARE MSG_TEXT CHARACTER(80) STATIC READONLY
                 INITIAL ('Good morning');
```

This use of the READONLY attribute provides storage optimization and protects variables from inadvertent modification.

12.2.2 The VALUE Attribute

A variable declared with the VALUE attribute does not require an address reference in storage; instead, the compiler can refer to it by value during execution.

When you give a variable the VALUE attribute, you must specify either GLOBALDEF or GLOBALREF. If you specify GLOBALDEF, you must use the INITIAL attribute to define a value for the variable.

For example, the VALUE attribute can be specified in the declaration of an external global symbol, as follows:

```
DECLARE REQUEST_CODE GLOBALDEF VALUE FIXED BINARY
   STATIC INITIAL (10);
```

The variable REQUEST_CODE in this example can be accessed in any external procedure that declares it with the attribute GLOBALREF.

When the VALUE attribute is used with the GLOBALDEF or GLOBALREF attributes, the following rules apply:

- The variable can have only one of the following data types:

  FIXED BINARY
  BIT (n) ALIGNED where n is less than 33

- The variable must be scalar.
- The value of the variable cannot be modified. Thus, it cannot be used as the target of an assignment statement or an input operation, nor can it be passed by reference in a procedure call. PL/I always creates a dummy argument for a variable with the VALUE attribute that is specified in an argument list.
- All declarations of the variable must specify the VALUE attribute.
- 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.

12.3 Obtaining Definitions for System Global Symbols

Within the OpenVMS 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 the following:

- 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 as follows:

```
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 Chapter 11.

13
Mailboxes

A mailbox is a virtual I/O 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, and examples of simple procedures that perform input and output to mailboxes.

Note that this chapter provides only information that is pertinent to mailbox I/O 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 Chapter 11.

13.1 Using Mailboxes

This section provides information on how the system controls the creation and use of mailboxes, and shows a typical use of mailboxes in an application.

13.1.1 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 you to create a mailbox that is automatically deleted when the image that created it completes execution.
- The privilege to create permanent mailboxes (PRMMBX) permits you to create a mailbox that continues to reside in system memory until it is specifically deleted.

In either case, when the system creates a mailbox, it defines a unique device with the name MBn (where n is a unit number) and equates this device name with the logical name specified by the program that created the mailbox.

The logical name of a temporary mailbox is placed in the group logical name table for the group of the creating process. The logical name of a permanent mailbox is placed in the system logical name table.

The process that creates a mailbox can define its protection; that is, it can control which users are allowed to write messages to the mailbox and which users are allowed to read messages from the mailbox.

13.1.2 Applications

A mailbox usually has only one reader and multiple writers. In a typical application, two or more program images would be executed concurrently in separate processes. One program, the controlling program, receives requests or messages from the other cooperating programs by way of the mailbox.

The controlling program takes the following actions:

1. It creates a temporary mailbox and gives it a logical name.
2. It associates a PL/I file constant with the mailbox by specifying the mailbox logical name in the TITLE option of an OPEN statement.
3. It executes a READ statement that initiates a read request.
4. When the READ statement is completed, it processes the data obtained from the mailbox.
5. It repeats steps 3 and 4 until no more data is written to the mailbox.
6. It issues a CLOSE statement to dissociate the file constant and delete the mailbox. The logical name for the mailbox is automatically deassigned when the mailbox is deleted.

Each cooperating program takes the following actions:

1. It associates a PL/I file constant with the mailbox by specifying the logical name of the mailbox in the TITLE option of an OPEN statement.
2. It executes WRITE or PUT statements that output data to the mailbox.
3. It continues to write to the mailbox until it no longer needs to send data or requests.
4. It executes a CLOSE statement to dissociate the PL/I file from the mailbox.

The following describes a typical application of mailbox communication between processes:

1. The controlling process creates the mailbox with the logical name PLI_MAILBOX and assigns it device name MBA99.
2. The controlling process opens the mailbox file with the following statement:

```
OPEN FILE (MFILE) INPUT RECORD
     TITLE('PLI_MAILBOX');
```

3. The controlling process reads and handles information from the mailbox continuously:

```
LOOP: READ FILE (MFILE) INTO(M_REC);
      .
      .
      .
      GOTO LOOP;
```

The mailbox remains available to other processes until the controlling process deletes the mailbox.

To write to the mailbox, a program does the following:

1. Opens the mailbox for output:

```
OPEN FILE (MAILB) OUTPUT RECORD
     TITLE('PLI_MAILBOX');
```

2. Writes messages to the mailbox:

```
WRITE FILE (MAILB) FROM (M_TEXT);
```

3. Closes the file when all messages have been sent:

```
CLOSE FILE (MAILB);
```

All processes writing to the mailbox must specify the TITLE defined by the process that created the mailbox, but it can specify its own file and record names.

13.1.3 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 I/O 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 counts all channels assigned to a mailbox; therefore, it knows when to delete the mailbox.

13.1.4 Effects of the CLOSE Statement

A CLOSE statement for a mailbox dissociates 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. You must invoke the Delete Mailbox system service (SYS$DELMBX) 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.

Note that in the context of reading a mailbox, an end-of-file does not necessarily mean that there is no more data; it only means that one channel has been deassigned. Thus, a program that is reading a mailbox must take end-of-file records into account and handle the ENDFILE condition accordingly. The manner in which the ENDFILE condition is handled depends on the type of I/O being performed, as described in Section 13.2.

13.2 Mailbox Input/Output

You can use either the stream I/O statements GET and PUT or the record I/O statements READ and WRITE to read from and write to mailboxes. The type of I/O you use depends on your application and the type of data that will be sent.

When you plan an application using mailboxes, you must also determine whether to use synchronous or asynchronous I/O operations. These types of operations are described in the following sections. Each involves special programming considerations.

13.2.1 Synchronous Input/Output

By default, all I/O operations to a mailbox are synchronous. This means that when an image executing in one process performs an output operation to a mailbox, the operation is not completed until an image being executed in another process reads the data from the mailbox. Similarly, when a program requests an input operation from a mailbox, control is not returned until an actual input operation is performed: if there is no data in the mailbox, the process must wait until another process writes data to the mailbox.

Example 13–1 shows a program that reads a mailbox synchronously. This program reads all data sent to a particular mailbox and copies all messages into a central log file. This procedure assumes that the mailbox PLI_MAILBOX already exists. For an example of a procedure that creates this mailbox, see Chapter 11.

The following notes are keyed to Example 13–1:

1 The procedure LOGGER declares the identifiers MAILFILE and OUTFILE with the FILE attribute.

2 The structure LOG_MESSAGE depicts the format of messages that are written to the mailbox. By a convention established for the application in this example, all programs in this application write messages with fields of these data types and lengths.

The first longword in the message is a type code. This is a convention used by OpenVMS system procedures that use mailboxes.

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 input loop of the procedure. This statement ensures that LOGGER will not be accidentally terminated if an ENDFILE condition is signaled when a program executes 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; LOGGER closes both files and returns.

Example 13–1 Synchronous Mailbox Input/Output

```
LOGGER: PROCEDURE;
    DECLARE (MAILFILE,OUTFILE) FILE;    /* 1  */
    DECLARE
        1 LOG_MESSAGE,                          /* 2  */
          2 TYPE FIXED BINARY(31),
          2 SYSTEM_TIME CHARACTER(25),
          2 REQUESTOR CHARACTER(15),
          2 STATUS FIXED BINARY(31);

    %REPLACE END_RUN BY -1;

    OPEN FILE(MAILFILE) RECORD INPUT SEQUENTIAL    /* 3  */
                TITLE ('PLI_MAILBOX');
    OPEN FILE(OUTFILE) PRINT TITLE('MAILTEST.OUT');   /* 4  */

    ON ENDFILE(MAILFILE) GOTO LOOP; /* Ignore end-of-file */   /* 5  */
LOOP:
    READ FILE(MAILFILE) INTO (LOG_MESSAGE);

    IF LOG_MESSAGE.TYPE = END_RUN     /* 6  */
    THEN
        GOTO FINISH;

    PUT FILE(OUTFILE) SKIP LIST(TYPE,
                SYSTEM_TIME,REQUESTOR,STATUS);
    GOTO LOOP;
FINISH:
    CLOSE FILE(MAILFILE), FILE(OUTFILE);   /* 7  */

    END LOGGER;
```

13.2.2 Asynchronous Input/Output

It is not always practical for a procedure that is reading a mailbox to wait until the mailbox has been written. To perform an I/O operation that is completed immediately, you must code a call to the Queue I/O Request system service (SYS$QIO). This service permits you to specify I/O functions that are not possible using PL/I statements.

Example 13–2 illustrates a procedure that uses the SYS$QIO system service to perform asynchronous I/O to a mailbox. This procedure, EMPTY_BOX, reads all of the messages in a mailbox. If the mailbox is empty, EMPTY_BOX displays a message to that effect.

The following notes are keyed to Example 13–2:

1 The system services SYS$ASSIGN, SYS$QIO, and SYS$WAITFR are declared.

2 The procedure includes $IODEF from PLI$STARLET.TLB, which defines symbol names for the I/O function codes.

3 The variable MESSAGE is the buffer into which the mailbox messages will be read.

4 The call to SYS$ASSIGN specifies the logical name of the mailbox, PLI_MAILBOX, and the variable MBXCHAN. SYS$ASSIGN returns the number of the channel.

5 In the call to SYS$QIO, the procedure specifies an event flag on which to wait for I/O completion and the channel number.

6 The next argument to SYS$QIO is the function code and its modifier, whose values are added to obtain the correct I/O function. The values of these names have the following meanings:

- The function code IO$_READVBLK is an instruction to read data.
- IO$M_NOW is a function modifier that specifies that control be returned to the calling program immediately.

7 The I/O status block argument is specified so that the status of the I/O operation can be determined and the length of the message read can be used. The missing arguments in this call are an AST routine address and an AST parameter.

8 The IO$_READVBLK function code requires the specification of the address of a message buffer and the size of the buffer. These are the last arguments specified in this call to SYS$QIO.

9 The procedure waits for the I/O to be completed.

10 EMPTY_BOX checks the status value in the I/O status block. If not successful, and if the unsuccessful status is not SS$_ENDOFFILE, the procedure exits. Otherwise, EMPTY_BOX displays the message and loops back to the beginning of the DO-group. If the status in the I/O status block was SS$_ENDOFFILE, the DO-group is not executed and the program is completed.

Example 13–2 Asynchronous Mailbox Input/Output

```
EMPTY_BOX: PROCEDURE OPTIONS(MAIN) RETURNS (FIXED BINARY(31));
    %INCLUDE LIB$GET_EF;
    %INCLUDE LIB$FREE_EF;
    %INCLUDE SYS$ASSIGN;    /* 1  */
    %INCLUDE SYS$QIO;
    %INCLUDE SYS$WAITFR;
    %INCLUDE $IODEF;        /* 2  */
    %INCLUDE $SSDEF;
    %INCLUDE $STSDEF;
    DECLARE MBXCHAN FIXED BINARY(15);
    DECLARE EFN FIXED BIN(31);
    DECLARE
        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);    /* 3  */
    STS$VALUE = SYS$ASSIGN('PLI_MAILBOX',MBXCHAN,,);    /* 4  */
    IF ^STS$SUCCESS
    THEN
        RETURN (STS$VALUE);
    /*
     * Get an event flag to use
     */
    STS$VALUE = LIB$GET_EF(EFN);
    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.
     */
    IO_STATUS.VALUE = 0;
    DO WHILE(IO_STATUS.VALUE ^= SS$_ENDOFFILE);
        STS$VALUE = SYS$QIO (
                        EFN,                        /* 5  */
                        MBXCHAN,
                        IO$_READVBLK | IO$M_NOW,    /* 6  */
                        IO_STATUS,,,                /* 7  */
                        MESSAGE,                    /* 8  */
                        LENGTH(MESSAGE),,,,);
        IF ^STS$SUCCESS
        THEN
            RETURN(STS$VALUE);
        STS$VALUE = SYS$WAITFR(EFN);    /* 9  */
        IF IO_STATUS.VALUE = SS$_ENDOFFILE
        THEN DO;
            PUT SKIP LIST('Mailbox empty');    /* 10 */
            RETURN(1);
            END;
        IF ^IO_SUCCESS
        THEN
            RETURN(IO_STATUS.VALUE);
```

(continued on next page)

Example 13–2 (Cont.) Asynchronous Mailbox Input/Output

```
        /*
         *  If successful read, fall through to here
         */
        PUT SKIP LIST(SUBSTR(MESSAGE,1,IO_STATUS.BYTES_TRANSFERRED),
                'status ',IO_STATUS.VALUE);
        END;

    /*
     * Release the event flag
     */
    STS$VALUE = LIB$FREE_EF(EFN);
    RETURN (STS$VALUE);

    END EMPTY_BOX;
```

14
Accessing Files on a Network

If your system supports DECnet facilities, and your computer is one of the nodes in a DECnet network, you can communicate with other nodes in the network by means of standard PL/I I/O 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 files were on your local system.
- Task-to-task communication lets you exchange data directly with a job that is being executed 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 details on using the DECnet facilities, see the *DECnet for OpenVMS Guide to Networking* or the *Guide to DECnet–VAX Networking*.

14.1 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 [MALCOLM] on the device DBA0: 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

For example:

```
OPEN FILE (NETFILE) SEQUENTIAL INPUT RECORD
      TITLE
('TULSA::DBB0:[MALCOLM]PLITEST.DAT');
```

This OPEN statement specifies the name of a file to be read from the node named TULSA.

If no file specification is present in the TITLE option, or if the TITLE option specifies a logical name, you can define a remote file. For example:

```
OPEN FILE(INFILE) INPUT RECORD SEQUENTIAL;
```

This OPEN statement refers to the logical name INFILE. The following DEFINE command equates this logical name with a remote file:

```
$ DEFINE INFILE TULSA::DBB0:[MALCOLM]PL1TEST.DAT
```

When the OPEN statement is executed, the run-time system associates the remote file with the PL/I file INFILE.

When you run a program that modifies a file on a remote node in a protected account, the file specification must contain an access control string. Each Digital operating system defines the format of an access control string. For an OpenVMS system, you specify the user name and password of the account whose file you are modifying. For example:

```
$ DEFINE INFILE -
$_TULSA""MALCOLM YES""::DBB0:[ENERGY]PL1TEST.DAT"
```

The user name MALCOLM and the password for this account (YES) are enclosed in quotation marks following the node name in the file specification. The extra quotation marks are required because the DCL command interpreter removes single pairs of quotation marks from lines. On the system at the node TULSA, the user MALCOLM must have access privileges to the account ENERGY.

The following file system functions are not available for processing remote files: keyed DELETE, WRITE, REWRITE FILE_ID_TO, FILE_ID_FROM, RECORD_ ID_TO, and RECORD_ID_FROM statements cannot be followed by a sequential operation, that is, an operation that does not specify a key.

14.2 Task-to-Task Communication

Network task-to-task communication lets a program running on one network node interact with a program running on another network node. The interaction is accomplished with PL/I I/O statements, but network connections themselves are transparent to the cooperating programs.

PL/I programs at remote locations can communicate over the network by the following mechanism:

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. For example:

   ```
   OPEN FILE (TASKFILE) RECORD OUTPUT
        TITLE ('HSTN"MALCOLM YES"::"TASK=LOGGER" ');
   ```

   This OPEN statement initiates task-to-task communication with the target node by specifying the task name LOGGER. The network program uses the default directory of user MALCOLM to locate the command file LOGGER.COM on the remote target node.

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 by the remote system. When the OPEN statement completes, communication between the two tasks can begin.

3. The command file LOGGER.COM must contain the command necessary to initiate the execution of the cooperating program, COPYTASK.

   ```
   $ RUN COPYTASK
   ```

   The network program submits the specified command file to the batch job queue on the target system.

4. The cooperating target task must complete the connection to the source task by executing an OPEN statement to open the file SYS$NET.

   ```
   OPEN FILE (NETFILE) RECORD
        SEQUENTIAL INPUT TITLE ('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.

Examples 14–1 and 14–2 illustrate PL/I programs that communicate across the network using synchronous I/O. The following notes are keyed to Example 14–1:

1 The procedure SOURCE_TASK is the program that initiates the request.

2 The UNDEFINEDFILE condition will be signaled if any error occurs that is associated with the logical link or connection. In the ON-unit, the procedure uses the ONCODE built-in function to obtain the error code, display the status value, and stop the program.

3 The OPEN statement associates the PL/I file TASKNAME with the task named LOGGER on the node named BOSTON. The network program uses the default directory for the account BEANS on the node and locates the command file LOGGER.COM. The file LOGGER.COM contains this command:

```
$ RUN TARGET
```

4 The procedure writes three messages from the structure TASK_MESSAGE. The first field is a binary value, and the second field a character-string text.

5 When the three messages have been written, the CLOSE statement closes the file to terminate the network connection.

Example 14–1 A PL/I Network Source Task

```
SOURCE_TASK: PROCEDURE;    /* 1  */

    DECLARE TASKNAME FILE;

    DECLARE 1 TASK_MESSAGE,
              2 NUMBER FIXED BINARY(31),
              2 TEXT CHARACTER(40) VARYING;
```

(continued on next page)

Example 14–1 (Cont.) A PL/I Network Source Task

```
ON UNDEFINEDFILE(TASKNAME) BEGIN;    /* 2  */
    PUT SKIP LIST('File error',ONCODE());
    STOP;
    END;
OPEN FILE(TASKNAME) SEQUENTIAL OUTPUT RECORD    /* 3  */
       TITLE('BOSTON"BEANS BAKED"::"TASK=LOGGER"');

NUMBER = 1;             /* 4  */
TEXT = 'first message';
WRITE FILE (TASKNAME) FROM (TASK_MESSAGE);
NUMBER = 2;
TEXT = 'second message';
WRITE FILE (TASKNAME) FROM (TASK_MESSAGE);
NUMBER = 3;
TEXT = 'third and last message';
WRITE FILE(TASKNAME) FROM (TASK_MESSAGE);

CLOSE FILE(TASKNAME);      /* 5  */
RETURN;

END;
```

The following notes are keyed to Example 14–2:

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 in 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 the file is opened successfully, 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 named 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.

Example 14–2 A PL/I Target Task

```
TARGET_TASK: PROCEDURE;   /* 1  */
    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(' ') (A(80));

    ON UNDEFINEDFILE(INFILE) GOTO FILE_ERROR; /* Network errors */  /* 2  */

    ON ENDFILE(INFILE) GOTO FINISH;     /* Normal completion */  /* 3  */

    OPEN FILE (INFILE) RECORD SEQUENTIAL INPUT
            TITLE ('SYS$NET');    /* Open SYS$NET */   /* 4  */
    OPEN FILE(OUTFILE) RECORD SEQUENTIAL OUTPUT
            TITLE('TASK.DAT');      /* Open output log file */

    LOOP:                                          /* 5  */
            READ FILE(INFILE) INTO (LOG_MESSAGE);
            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;
```

15
Storage Allocation

This chapter describes the following topics:

- Program sections, which are groupings of data made by the compiler and used by the linker
- The addressability of variables

Refer to the *PL/I for OpenVMS Systems Reference Manual* for information on storage classes.

15.1 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 data is grouped according to its attributes—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 allocates virtual memory for the image, it groups program sections that have similar attributes.

15.1.1 Attributes of Program Sections

Table 15–1 lists the attributes that can be applied to program sections.

Table 15–1 Program Section Attributes

Attribute[1]	Meaning
PIC or NOPIC	The program section or data it refers to does not depend on any specific virtual memory location (PIC), or the program section depends on one or more virtual memory locations (NOPIC).
CON or OVR	The program section will be concatenated with other program sections with the same name (CON), or will be overlaid on the same memory locations (OVR).
REL or ABS	The data in the program section must be relocated to a virtual memory address (REL), or does not occupy virtual memory (ABS).
GBL or LCL	The program section contains definitions for symbols that are shared with other program sections or modules (GBL), or are local to the current program section (LCL).

[1]This column lists pairs of conflicting attributes.

(continued on next page)

Table 15–1 (Cont.) Program Section Attributes

Attribute[1]	Meaning
EXE or NOEXE	The program section contains executable code (EXE), or does not contain executable code (NOEXE).
WRT or NOWRT	The program section contains data that can be modified (WRT), or data that cannot be modified (NOWRT).
RD or NORD	These attributes are not currently used.
SHR or NOSHR	The program section can be shared in memory (SHR), or cannot be shared in memory (NOSHR).

[1]This column lists pairs of conflicting attributes.

15.1.2 Program Sections Created by PL/I

PL/I creates the following program sections for every program:

- $CODE—contains all executable code and constant data.
- $DATA—contains all internal static variables.

PL/I also creates additional program sections for external variables and global symbols. Table 15–2 summarizes the program sections that PL/I creates for variables declared with different storage class attributes.

Table 15–2 Program Sections for PL/I Variables

Storage Class Attributes	Program Section Name[1]	Program Section Attributes
EXTERNAL CONTROLLED	*name*	PIC, OVR, REL, GBL, NOSHR, NOEXE, RD, WRT
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
Not user-specified	$ADDRESS_DATA	PIC, CON, REL, LCL, NOSHR, NOEXE, RD, NOWRT

[1]Here, *name* is the identifier of the variable declared with the specified attribute, and *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.

15.1.3 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 section represents a separate program section; each procedure that declares the common section with the same name can access the same variable.

A PL/I external variable called XYZ therefore corresponds to a FORTRAN common section called XYZ. The following examples illustrate PL/I procedures and FORTRAN procedures that share data.

STRING.PLI

```
STRING: PROCEDURE OPTIONS(MAIN);
DECLARE XYZ EXTERNAL CHARACTER(20),
     PRSTRING ENTRY;
     XYZ = 'THIS IS A STRING';
     CALL PRSTRING;
END;
```

PRSTRING.FOR

```
          SUBROUTINE PRSTRING

          CHARACTER*20 STRING
          COMMON /XYZ/ STRING
          WRITE (6,20) STRING
 20       FORMAT (' ',A20)
          RETURN
          END
```

In this example, the PL/I external variable XYZ corresponds to the FORTRAN common section named XYZ. The FORTRAN procedure displays the data in the common section.

To share more than one variable in a program section with a FORTRAN program, the PL/I variables must be declared within a structure. For example:

NUMBERS.PLI

```
NUMBERS: PROCEDURE;

DECLARE 1 NUMBERS EXTERNAL,
          2 FIRST FIXED(31),
          2 SECOND FIXED(31),
          2 THIRD FIXED(31),
        FNUM ENTRY;

        FIRST = 1;
        SECOND = 2;
        THIRD = 3;

        CALL FNUM;
END;
```

FNUM.FOR

```
          SUBROUTINE FNUM

          INTEGER*4 INUM,JNUM,KNUM
          COMMON /NUMBERS/ INUM,JNUM,KNUM
          WRITE (6,10) (INUM,JNUM,KNUM)
 10       FORMAT (3I8)
          RETURN
          END
```

In this example, the fixed binary variables declared in the PL/I external structure NUMBERS correspond to the FORTRAN INTEGER*4 variables in the common section of the same name. Note that in a FORTRAN common section, all variables must be either integers or character strings. Variables of different data types cannot be grouped into the same common section.

15.2 Addressability

Variables are either addressable or nonaddressable. In some contexts, such as in argument lists of certain built-in functions, a variable must be addressable. A variable is addressable if it has all of the following properties:

- It is not suitable for bit-string overlay defining; that is, it does not consist entirely of unaligned bit data.
- It is not an unconnected array (typically a member of an array of structures).
- It is not declared with the VALUE attribute.

These rules ensure that the variable can occupy contiguous storage beginning on a byte boundary. (Note that constants are never addressable in PL/I.)

A

PL/I Messages

This appendix describes the messages produced by the following:

- The PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compilers
- The OpenVMS run-time system
- The CRX (%CRX messages issued during use of the Common Data Dictionary)

The description of each message gives the severity, followed by additional explanatory text and suggested action. Compiler messages with severities of Error or Fatal require that you recompile the program after correcting the source text.

A.1 Compiler Messages

The diagnostic messages produced by the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP compilers are listed in this section, alphabetized by identification.

ADDRARG, The argument of ADDR must be a byte addressable reference.

Error: The argument of the ADDR built-in function is not on a byte boundary. For example, this error occurs if the argument is an unaligned bit string.

User Action: Verify that the correct argument is specified for the ADDR built-in function. If a bit-string variable is correctly specified, check that the declaration of the variable has the ALIGNED attribute.

ADDRNOTREF, The argument of ADDR must be a reference to a variable.

Error: The ADDR built-in function specifies an argument that is a constant or a global symbol.

User Action: Verify that the correct argument is specified for the ADDR built-in function and that the argument is not a constant.

AGGMISMAT, The source and target of an assignment are aggregates that do not match as required by the language rules.

Error: An assignment statement assigning the value of one array to another or one structure to another references arrays or structures that are not identical. This form of assignment is valid only when arrays with the same data type, number of dimensions, and extents are used, and when structures with the same hierarchy and data types are used.

User Action: Verify that the references in the assignment statement refer to the correct aggregates. Use separate assignment statements if the source and target of an assignment are aggregates that do not match.

ALIGNARRAY, The CDD description for array item *entity* contains the ALIGNED attribute. ALIGNED is being ignored by PL/I.

Informational: PL/I for OpenVMS VAX does not support the ALIGNED attribute on arrays, so if it is specified in the CDD it is ignored.

User Action: No action is necessary.

ALIGNED, *Entity* has been declared with the ALIGNED attribute. Only BIT or CHARACTER string variables can be declared ALIGNED.

Error: The ALIGNED attribute is specified with a conflicting attribute.

User Action: Correct the declaration of the variable.

AMBIGREF, This statement contains an ambiguous reference to *entity*.

Error: This error is produced when more than one structure contains a member with the same identifier name, and the name is referenced without a structure qualifier.

User Action: Determine the member name that should be referenced and correct the reference by including a structure qualifier of the form name1.name2.name3 . . . in the statement.

ANYCNOTSTAR, The ANY and CHARACTER attributes can only be used together if the CHARACTER attribute is CHARACTER(*).

Warning: The ANY and CHARACTER attributes can be used together only in the parameter declaration ANY CHARACTER(*). Fixed-length ANY CHARACTER declarations are not allowed.

User Action: Correct the declaration to specify the length as an asterisk.

ARGCVRT, Implicit conversion. A procedure argument, *entity*, has been converted to the parameter type *entity*.

Warning: The data type of the indicated argument does not match the data type of the corresponding parameter descriptor, and the PL/I compiler has converted the argument to the data type of the parameter. 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 argument, 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.

ARGLEQ253, A procedure reference contains more than 253 arguments.

Error: A CALL statement or a function reference specifies an argument list with more than 253 arguments.

User Action: Examine the argument list to see if there is a syntax error. If the list is correct, simplify the program so that no procedure requires more than 253 arguments.

ARGOMIT, An argument can be omitted with the ,, notation only when the called procedure is declared as OPTIONS(VARIABLE) or when the formal parameter is declared with the OPTIONAL attribute.

Error: An argument list in a procedure invocation contains null arguments, for example (a,,b).

User Action: Verify that the argument list in the procedure invocation specifies all arguments that are required. If the procedure accepts default arguments, declare the formal parameters with the OPTIONAL attribute.

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.

ARRAYOVFL, FIXEDOVERFLOW occurred in calculating the multipliers or virtual origin of the array *entity*.

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, *entity* has been converted to an *entity* 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 *entity*.

Error: Something is missing in a declaration.

User Action: Correct the declaration.

BADAGGARG, *Entity* is an invalid array, structure, or area argument. Such an argument must be a variable reference, must not be enclosed in parentheses, and must exactly match the corresponding parameter.

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.

BADALLOCN, The argument of ALLOCATION must be a reference to a CONTROLLED variable.

Error: The ALLOCATION built-in function returns the number of generations of a variable and is used only with CONTROLLED variables. A variable with a storage class other than CONTROLLED was given as the argument of the ALLOCATION built-in function.

User Action: Supply a valid argument and recompile the program.

BADANYARG, The procedure argument, *entity*, 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.

BADARG, The procedure argument, *entity*, is not valid for conversion to the *entity* parameter type.

Error: The indicated reference or expression specified in an argument cannot be converted to the data type of the corresponding parameter.

User Action: Determine whether the parameter descriptor was correctly specified or if the argument is correct, and change either the parameter descriptor or the argument list.

BADARITH, The noncomputational value, *entity*, has been used in a context requiring an arithmetic value.

Error: The indicated expression or reference has a data type that cannot be converted to an arithmetic expression, yet has been specified in a context that requires an arithmetic expression.

User Action: Correct the expression so that it specifies an arithmetic data type or a data type that can be converted to arithmetic.

BADASSIGN, The source operand, *entity*, of an assignment is invalid for conversion to the *entity* target.

Error: The variable or expression on the right side of the assignment statement has a data type that is incompatible with the data type of the target.

User Action: Correct the expression so that it specifies a data type that can be converted to the data type of the target. If the target is an array variable, or a member of a dimensioned structure, be sure that the array reference contains a valid subscript.

BADATATYPE, An expected *entity* value was not found. One of the values in this statement has a data type that is inconsistent with the context in which the value is used.

Error: A data item of one type has been specified in a context where an item of another data type is required. For example, this error occurs if the target

label on the GOTO statement is not a label but a variable declared with other data type attributes.

User Action: Verify the data type of the item, and correct the declaration of the variable.

BADBASE, The CDD description for structure item *entity* specifies an incompatible base.

Error: PL/I supports only decimal or binary numbers. The Common Data Dictionary structure item *entity* is neither decimal nor binary.

User Action: Change the Common Data Dictionary description to an appropriate data type.

BADBIT, The noncomputational value, *entity*, has been used in a context requiring a bit-string value.

Error: The indicated name or expression has a data type that cannot be converted to a bit string.

User Action: Correct the expression or the reference so that it has a data type that can be converted to a bit string.

BADBITCON, A bit-string constant contains an invalid digit or the digit following the B is not 1, 2, 3, or 4.

Error: A bit-string constant is incorrectly specified.

User Action: Determine the correct base for the bit-string constant, and correct the specification.

BADCHAR, A noncomputational value, *entity*, has been used in a context requiring a character-string value.

Error: The indicated variable or expression has a data type that cannot be converted to a character string, but is used in a context that would require such conversion.

User Action: Correct the expression so that it specifies a character-string data type or a data type that can be converted to character.

BADCLATTR, *Entity* is declared with duplicate or conflicting attributes. *Entity* conflicts with *entity*.

Error: This error occurs when conflicting attributes of any type are specified. Two examples of errors that produce this message are as follows:

- File description attributes are specified with data type attributes, or are specified for file variables or file parameters.
- 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 is the incorrect attribute of the name, and remove it from the declaration.

BADCLSLABL, The closure label in this statement does not match the label prefix of the containing DO, SELECT, BEGIN, or PROCEDURE block.

Error: Multiple closure is not permitted in PL/I for OpenVMS VAX. Each DO, BEGIN, PROCEDURE, and SELECT 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, PROCEDURE, or SELECT statement that does not already have a corresponding END statement.

BADCMPAREA, AREAs cannot be compared using relational operators.

Error: An AREA variable cannot be used in an expression with a relational operator.

User Action: Change the program to avoid using AREA variables in relational expressions.

BADCOMPARE, Invalid comparison. The operands of relational operators must both be arithmetic values, string values, or compatible noncomputational items. Noncomputational data other than AREAs can be compared only 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 that tests something 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, *entity*, 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.

BADDEFBAS, The base reference specified for the DEFINED variable *entity* is not a connected variable reference.

Error: A variable is declared with the DEFINED attribute, and the variable specified in the DEFINED attribute is a variable whose storage is unconnected.

User Action: Correct the variable declaration so that it refers to a variable whose storage is connected.

BADELEREF, The refer element *entity* references storage in the containing structure.

Error: The refer element cannot reference any storage in the structure containing the refer element.

User Action: Change the refer element.

BADENVAL, Invalid argument in an ENVIRONMENT option. An *entity* was not found where expected.

Error: An ENVIRONMENT option requires 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.

BADEXTRACT, The first argument of INT or POSINT is an array, structure, or named constant.

Error: The first element of INT or POSINT must be a reference to a scalar variable or expression. It cannot be an array, structure, or named constant. It can, however, be an array member or an elementary structure member.

User Action: Make certain that the first element of INT or POSINT is a legal argument.

BADFILATTR, This statement contains inconsistent file description attributes or options that conflict with those attributes.

Error: The attributes specified for a file constant in a DECLARE or OPEN statement are incompatible; for example, both the STREAM and the UPDATE attributes are specified. This error also occurs when conflicting ENVIRONMENT options are specified for a file.

User Action: Determine which attributes or options are in conflict, and correct the statement.

BADFMTLABL, *Entity* is not the label of a FORMAT statement.

Error: An R format item in a format specification list for a GET or PUT statement contains a reference to a name that is not the label of a FORMAT statement.

User Action: Verify that the label matches the label on a valid FORMAT statement, and correct the statement.

BADFREETAR, A FREE statement must free a nonmember BASED or CONTROLLED variable.

Error: A FREE statement specifies a variable that is not a BASED variable, a CONTROLLED variable, or a variable that is a member of a structure.

User Action: Correct the reference in the FREE statement.

BADINITVAL, *Entity* has been declared with an INITIAL attribute. An INITIAL attribute cannot be specified for variables of this storage class.

Warning: The INITIAL attribute is specified for a defined variable or for a parameter.

User Action: Ensure that the variable has the AUTOMATIC, STATIC, BASED, CONTROLLED, or GLOBALDEF attribute. The INITIAL attribute is invalid for all other storage classes.

BADINT, The noncomputational value, *entity*, has been used in a context requiring an integer value.

Error: The indicated expression or reference has a data type that cannot be converted to an integer, yet has been used in a context that requires an integer.

User Action: Correct the reference so that it specifies an integer.

BADKEYARG, The keyword argument *entity* does not match any formal argument name for this preprocessor procedure.

Error: A keyword specified in a preprocessor procedure with the STATEMENT option does not match any of the parameters.

User Action: Check the spelling of the keyword and recompile.

BADLABSUB, The label index in a LEAVE statement must be an integer constant.

Error: The label can be a label constant or a subscripted label constant, but the subscript must be specified with an integer constant.

User Action: If the label constant is a subscripted label constant, ensure that the subscript is an integer constant.

BADLEAVE, The LEAVE statement must be contained by a DO group in this block.

Error: A LEAVE statement must be contained in a DO-group, which can be nested. However, all DO-groups containing the LEAVE statement must be in the same block.

User Action: Ensure that the LEAVE statement is contained within a DO-group and that they are both contained in the same block.

BADLEFTSID, One of the targets of this assignment is not a variable or pseudovariable reference.

Error: The left side of an assignment statement contains a constant or an invalid reference.

User Action: If the target of the assignment is a variable, ensure that it is properly declared. If the target is a function reference, it must be one of the PL/I built-in functions that are valid as pseudovariables.

BADLIKEDCL, A variable declared with the LIKE attribute references another variable declared circularly LIKE itself.

Error: The LIKE attribute can reference major or minor structures known to the current block. But the LIKE attribute cannot directly or indirectly reference a structure containing itself.

User Action: If the LIKE attribute has already been used once, it may be necessary to specifically declare the members of the structure. Frequently used structure declarations can be entered in the PL/I for OpenVMS VAX Common Data Dictionary and then used repeatedly in programs.

BADLIKEREF, A variable declared with the LIKE attribute must reference a legal structure.

Error: The referenced structure contains errors that prohibit it from being recognized as a legal structure. Other error messages probably indicate the source of the error in the referenced structure.

User Action: Determine the errors in the referenced structure and correct them.

BADLIKEVAR, *Entity*, which has been declared with the LIKE attribute, is not a structure variable.

Error: The LIKE attribute can be applied only to major and minor structures that are known to the current block.

User Action: Check if the referenced structure is contained in the current block. If it is, determine whether it meets the syntax requirements of the LIKE attribute.

BADMEMBER, *Entity* has been declared with the MEMBER attribute, but it is not a structure member.

Error: The MEMBER keyword can be used only to denote that an item is a member of a structure. It cannot be used to force an entity that is not a structure member to have the MEMBER attribute.

User Action: Make the entity being declared into a structure member or remove the MEMBER keyword.

BADOPTVAR, *Entity* is declared OPTIONS(VARIABLE). Its formal parameters cannot be declared with the OPTIONAL, TRUNCATE, or LIST attributes.

Warning: OPTIONS(VARIABLE) procedures cannot have parameters declared with the OPTIONAL, TRUNCATE, or LIST attributes, because OPTIONS(VARIABLE) implies all of these.

User Action: Remove the OPTIONS(VARIABLE) attribute from the entry declaration.

BADOUTER, *Entity* is declared outside of a procedure. It cannot be declared with the AUTOMATIC storage class. STATIC has been assumed.

Warning: A variable can be declared outside of a procedure, but the variable must have an explicit storage class, which must not be AUTOMATIC. Storage class STATIC is assumed by default.

User Action: Declare the variable with an explicit storage class.

BADPARAM, *Entity* has been declared with the PARAMETER attribute, but it does not appear in any parameter list of this routine.

Error: The PARAMETER attribute was used for a declaration of something other than a parameter.

User Action: Use the entity being declared in a parameter list so that it really is a parameter, or remove the PARAMETER attribute from the declaration.

BADPAREN, This statement contains unbalanced parentheses.

Error: A statement contains different numbers of open parentheses and closed parentheses.

User Action: Verify the syntax of the statement to determine where a parenthesis is needed, and correct the statement.

BADPERSTMT, Invalid syntax in a preprocessor statement.

Error: A preprocessor statement has not been correctly specified. The format for a preprocessor statement is as follows:

```
%[label:] STATEMENT;
```

User Action: Verify the syntax and recompile the program.

BADPICTURE, *Entity* is an invalid picture.

Error: The string specified in a picture specification contains a character that is not a valid picture character, specifies an iteration factor for a character that must not be repeated, or contains an iteration factor that is not correctly specified.

User Action: Verify and correct the picture.

BADREFMEM, The member *entity* contains a REFER option and precedes the refer object.

Error: The REFER option has been specified in the program before the refer object has been declared.

User Action: Move the refer object declaration so that it precedes all references to it in the structure.

BADREFSTR, The non-based structure *entity* has a member that contains a REFER option.

Error: The REFER option can be used only with structures declared with the BASED attribute.

User Action: Declare the structure BASED, or remove the REFER option.

BADREPT, An incorrect repetition factor has been specified. A repetition factor of 1 has been supplied.

Warning: The compiler was unable to replicate the specified string because the replication factor was improperly specified.

User Action: Verify that the syntax is correct; the string should be enclosed in apostrophes and the replication factor enclosed in parentheses.

BADRETVAL, The value, *entity*, in a return statement is not valid for conversion to the *entity* 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, *Entity* 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 and remove the number preceding the variable name.

BADSTRUCT, *Entity* has been declared with the STRUCTURE attribute, but it is not a structure.

Error: The STRUCTURE keyword can be used only to indicate that an item is a structure. It cannot be used to force a non-structure entity to have the structure attribute.

User Action: Make the entity being declared into a structure or remove the STRUCTURE keyword.

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. This line is the first incorrect line.

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.

BADTRUNCATE, *Entity* is a multi-positional parameter. It cannot be declared with the TRUNCATE attribute.

Error: The TRUNCATE attribute is not supported for multi-positional parameters.

User Action: Remove the TRUNCATE attribute from the declaration.

BADTYPEDCL, A variable declared with the TYPE attribute *entity* references another variable declared circularly like itself.

Error: The TYPE attribute can reference variables known to the current block. But the TYPE attribute cannot directly or indirectly reference a variable whose type depends on the type of the current variable.

User Action: If the TYPE attribute has already been used once, it may be necessary to specifically declare the variable.

BADTYPEREF, A variable declared with the TYPE attribute *entity* must reference a legal structure.

Error: The referenced structure contains errors that prohibit it from being recognized as a legal structure. Other error messages probably indicate the source of the error in the referenced structure.

User Action: Determine the errors in the referenced structure and correct them.

BADUNION, *Entity* has the attribute UNION but does not have a level number or has no members.

Error: A variable declared with the UNION attribute must be a major or minor structure with level numbers. A reference to one member of a union refers to the storage occupied by all members of the union, that is, to all members having the next higher-level number. Therefore, level numbers locate storage for union members.

User Action: Include level numbers in the declaration of a union. Make certain that the UNION attribute refers to existing higher-level numbers.

BADUNSPREF, The argument of UNSPEC must be a reference to a scalar variable or a reference to an element of an array or a structure.

Error: The UNSPEC built-in function is used incorrectly.

User Action: Correct the reference to UNSPEC. Its argument cannot be a constant or a structure name.

BADVALUSE, An expected *entity* 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.

BASENOTAREA, The reference specified as the base area for the offset variable *entity* is not a reference to an area variable.

Error: The data type of the variable specified in the OFFSET (reference) attribute is not AREA.

User Action: If the reference is correct, make sure that its declaration contains the attribute AREA. Otherwise, specify a variable that is an area.

BASENOTLOC, The reference in the BASED attribute specified for the variable *entity* is not a reference to a locator variable.

Error: The expression specified in the BASED attribute for a BASED variable or in the SET option of an ALLOCATE statement is not a reference to a POINTER or OFFSET variable. The use of the BASED variable in the context of this message requires that the value be a locator variable that can be written.

User Action: Change the BASED expression or the SET option to be a reference to a locator variable.

BIFARGCNT, A built-in function has been referenced with the wrong number of arguments.

Error: Too many or not enough arguments are specified in a reference to a built-in function.

User Action: Verify the argument list required by the built-in function, and correct the statement.

BIFLTSCAL, A built-in function that produces a floating-point result cannot specify a scale factor.

Error: A built-in function that returns either a floating-point or fixed-point result, depending on its arguments, specifies a scale factor for an argument that is floating point.

User Action: Correct the argument list, omitting the scale factor.

BIGINT, The integer value *entity* is too big for the context in which it occurs.

Error: An integer constant or an integer value that can be computed at compile time is too large in magnitude. For example, in SUBSTR(S,I,1024*124), the size argument is too large.

User Action: Determine the valid range of values for the context, and correct the source program.

BIGPICTURE, *Entity* results in a compiled picture that exceeds the implementation's limit. Reduce the size of the picture and recompile.

Error: A picture specification is too complex or contains more than 255 characters.

User Action: Correct the picture.

BITNOTBIN, Implicit conversion. A bit string, *entity*, has been used as the first operand of a FIXED, FLOAT, or DECIMAL built-in function.

Warning: The first operand of the built-in functions listed must be arithmetic, but a bit-string argument was specified. The compiler converted the bit-string argument to binary.

User Action: To avoid this message when you want the compiler to convert the bit-string expression to the appropriate arithmetic data type, use the BINARY built-in function, followed by a FIXED, DECIMAL, or FLOAT built-in function, if necessary. You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

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.

BLAWHEZER, The CDD description for structure item *entity* contains the Blank When Zero attribute. Blank When Zero is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Blank When Zero attribute.

User Action: None.

CDDERROR, Common Data Dictionary description extraction condition for pathname *entity*.

Informational: The PL/I compiler is in the process of extracting a data definition from the Common Data Dictionary.

User Action: See the accompanying messages for more information.

CDDTOOBIG, The attributes for some member of the Common Data Dictionary record description *entity* exceed the implementation's limit for member complexity.

Error: A member of the Common Data Dictionary record description has too many attributes and has created a program that is too large.

User Action: Change the Common Data Dictionary description to make the field description smaller.

CDDTOODEEP, The attributes for the Common Data Dictionary record description *entity* exceed the implementation's limit for record complexity.

Error: The Common Data Dictionary record descriptions are nested too deeply.

User Action: Change the Common Data Dictionary description to reduce the level of nesting in the record description.

CIRCDECL, The declaration of *entity* is circular. Some reference or expression in this declaration depends on the declaration itself.

Error: The compiler cannot resolve a reference, as in DECLARE P POINTER BASED(P).

User Action: Correct the reference so that its definition depends on some other variable.

CMPLXDOPE, The dope vector required for the argument *entity* 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 *entity* condition.

Error: Only the I/O condition names and the VAXCONDITION condition name can specify values.

User Action: Correct the ON condition name in the statement.

COLMAJOR, The CDD description for structure item *entity* specifies that it is a column-major array.

Error: PL/I only supports row-major arrays.

User Action: Change the Common Data Dictionary description to specify a row-major array.

CONFLATTR, Attributes declared for *entity* 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 invalid, and correct the statement.

CONFLOPT, The *entity* options *entity* and *entity* conflict.

Error: The specified recording locking options conflict with each other.

User Action: Remove some of the options.

CONPREC, The precision arguments of BINARY, DECIMAL, FIXED, FLOAT, DIVIDE, ADD, and MULTIPLY 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 an integer constant.

CONSTCOND, A condition occurred while an expression with constant operands was being evaluated.

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 which 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.

CONSTCVT, An ERROR occurred during the conversion of the constant *entity* to the context in which it is used.

Error: A constant is either invalid for conversion to another data type (for example, the string 'XXX' cannot be converted to arithmetic), or the FIXEDOVERFLOW or OVERFLOW condition occurred during the conversion.

User Action: Correct the constant.

CVTBIFSCAL, The scale factor specified in a conversion built-in function does not lie in the range *entity*:31.

Error: A conversion built-in function, for example, DECIMAL, is invoked with a scale-factor argument that is not a valid precision.

User Action: Determine the correct range of precision for the indicated data type and specify a value in that range in the argument list for the built-in function.

DCLENGTH, *Entity* has been declared with a length or size less than 0 or greater than the maximum for its data type.

Error: The variable is incorrectly declared.

User Action: Determine the correct range of values for the data type of the indicated variable and correct its declaration.

DCLEXPRES, An expression or reference contained within this DECLARE statement is excessively complex. Reduce the complexity and recompile.

Fatal: The compiler cannot interpret the statement.

User Action: Rewrite the declaration using two or more statements.

DCLTOOLONG, The total number of declarations, parameter descriptors, and returns descriptors in this DECLARE statement exceeds the implementation's limit.

Fatal: The compiler cannot interpret the statement.

User Action: Simplify the statement. If it is a DECLARE statement, place the declarations in different statements. If the statement contains a parameter descriptor or returns descriptor, simplify the procedure calling sequences.

DECDIVSCA, Use of the division operator resulted in a negative scale for the result. A result scale of zero has been used instead.

Warning: The division operator (/), when used to divide fixed decimal operands, yielded an expression that would have a negative scale factor in full PL/I. The PL/I for OpenVMS VAX compiler does not allow negative scale factors for fixed-point decimal numbers.

User Action: Use the DIVIDE built-in function to divide the fixed-point operands, and specify the scale of the result.

DEFBASCLA, The base reference specified for the DEFINED variable *entity* is itself DEFINED or BASED.

Warning: The PL/I for OpenVMS VAX language does not allow a DEFINED variable to be defined on a variable that is either BASED or DEFINED.

User Action: If you intended to define the variable on a BASED or DEFINED variable, you need do nothing; the results are likely to work as you expect.

DEFDATATYP, The undeclared name *entity* has been declared as a FIXED BINARY variable in the containing procedure.

Warning: A name that is not declared or that is not a label has been referenced; the compiler gives the attributes FIXED BINARY by default.

User Action: Check that the reference is correctly spelled; if it is not, correct the spelling. If the variable is not declared, declare it with the appropriate attributes for its use.

DESCRIBIF, Invalid use of the DESCRIPTOR built-in function.

Error: The DESCRIPTOR built-in function can be used in an argument list only to specify that an argument be passed by descriptor to a non-PL/I procedure; the corresponding parameter must be declared with the attributes ANY and VALUE. This message indicates that you are using the built-in function in an inappropriate context. Note that by default the PL/I compiler passes character strings and arrays by descriptor, so you need not specify the manner in which they are passed.

This message is also issued when the DESCRIPTOR built-in function is specified with more than one argument or with an argument that is noncomputational, pictured, or an array of noncomputational or pictured data.

User Action: Remove the reference to the built-in function from the statement.

DICTABORT, %DICTIONARY processing of the Common Data Dictionary record description *entity* aborted.

Error: The PL/I compiler is unable to process the Common Data Dictionary record description.

User Action: See the accompanying messages for further information.

DIVSCALE, Use of the division operator produced a result whose scale factor would exceed the implementation limit.

Error: The resulting scale factor must be in the range –31 through 31.

User Action: Use the DIVIDE built-in function to control the scale factor used in the division operation, or change the scale factors of the values used in the division operation.

DUMMYARG, A dummy argument has been created for *entity*, because it does not exactly match the *entity* parameter.

Warning: The compiler converted the argument to the data type of the corresponding parameter, and placed the result in a dummy argument. Thus, it passes 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 be 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 procedure may modify it, correct the declaration of the argument or the parameter descriptor or parameter list for the corresponding parameter.

DUPATTR, Duplicate attribute in list near *entity*.

Error: An attribute was specified more than once for the declaration.

User Action: Change the list to specify attributes only once for each entity being declared.

DUPDCL, This statement contains a duplicate declaration of *entity*.

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 incorrect 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 or the subscript and all references to it.

DUPOPTN, This statement contains duplicate, missing, or conflicting 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 incorrect one, and delete it from the statement.

DUPPRESCA, Multiple precisions or scale factors have been specified for this variable.

Error: A variable can only have one precision and scale factor specified. For example, declarations such as FIXED(31) BINARY(15) are not valid.

User Action: Remove the duplicate specifications.

DUPSIGN, *Entity* contains multiple sign symbols.

Error: A picture specification contains more than one plus or minus sign symbol.

User Action: Correct the picture so that it contains only a single sign.

EMPTYARG, *Entity* has been referenced with an argument list that 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.

ENDGIVEN, An END statement has been supplied to close a DO-group, SELECT-group, begin block, or procedure.

Warning: The compiler inserted an END statement in the file.

User Action: The listing file, if any, indicates the END statement that was inserted by PL/I. Verify that PL/I placed the END statement in the correct position. This message is informational, and the program may be correct; however, you should correct the source program and insert the END statement.

ENTRYGIVEN, *Entity* has been declared with a RETURNS attribute but no ENTRY attribute. An ENTRY attribute has been supplied.

Warning: The compiler supplied the ENTRY attribute for a name that has the RETURNS attribute.

User Action: Specify the ENTRY attribute on the declaration of the external entry to avoid this message. This message is a warning, and the program will be executed correctly.

ENTRYVALUE, An internal procedure is being passed by value. Uplevel references to AUTOMATIC variables and PARAMETERS will be invalid.

Warning: If an internal procedure is passed by value, it cannot make uplevel references to automatic variables or parameters because the entry's frame pointer will not be passed along with the address of the entry.

User Action: Change the declaration of the parameter to entry, or pass an external procedure instead of an internal one.

ENTYPEDEF, The variable *entity* TYPE reference is an entry point which is not allowed.

Error: An entry TYPE reference is not allowed.

User Action: Declare the variable with explicit attributes.

ENVSYN, Invalid syntax or value in ENVIRONMENT option.

Error: An ENVIRONMENT option is specified with an expression of a data type that is not valid for the option.

User Action: Determine the data type required by the option, and correct the expression.

EXTRATEXT, The source text contains extraneous data. Check for excess END statements, unbalanced apostrophes, and unbalanced /* */.

Error: The source file contains data following the last END statement, or following a string or comment.

User Action: Examine the listing file to locate the error. If there are too many END statements, check that all PROCEDURE and BEGIN statements are specified in the correct sequence. Correct the source program.

FIXBPREC, The precision specified for *entity* exceeds the implementation's limit of FIXED BINARY(31). The maximum precision of 31 has been supplied. Use FIXED DECIMAL for larger values.

Warning: The maximum precision for fixed-point binary variables is 31; the compiler changed a larger value to the maximum.

User Action: Correct the declaration of the variable so that it does not specify a precision greater than 31.

FIXBSCALE, The scale factor q specified for *entity* is not in the range -31<=q<=p, where p is the variable's precision. The scale factor has been set to zero.

Warning: Fixed-point binary numbers may have a scale factor within the range –31 to 31, but the scale factor must not be greater than the specified precision.

User Action: Adjust the scale factor so that it is less than the specified precision.

FIXDPREC, The precision specified for *entity* exceeds the implementation's limit of FIXED DECIMAL(31). The maximum precision of 31 has been supplied.

Warning: The compiler changed the precision of the fixed-point decimal variable.

User Action: Correct the declaration of the variable so that it does not specify a precision greater than 31.

FIXDPRECZERO, The precision specified for a fixed decimal is zero which is not in the implementation defined range of 1-31.

Error: The precision of a fixed-point decimal must be in the implementation defined range of 1-31. This error indicates that a fixed-point decimal variable was declared with a precision of zero, or one of the built-in functions specified a fixed decimal precision of zero.

User Action: Either correct the declaration of the variable so that it does not specify a precision of zero or correct the precision argument in the built-in function reference that caused the error.

FLTBPREC, The precision specified for *entity* exceeds the implementation's limit of FLOAT BINARY(*entity*). The maximum precision of *entity* 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 *entity* exceeds the implementation's limit of FLOAT DECIMAL(*entity*). The maximum precision of *entity* 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 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 less.

ILLABEL, Duplicate or out of order label for a GOTO statement with a label array reference.

Error: A label with an asterisk subscript (i.e., label_name(*)) is out of order or is a duplicate.

User Action: Remove the condition causing the error.

ILLIKEREF, The variable *entity* is declared with the LIKE attribute and references a structure that contains a REFER option.

Error: The REFER option conflicts with the LIKE attribute because the REFER option dynamically remaps storage for a structure.

User Action: Rename the variable or remove either the LIKE attribute or the REFER option from the declaration.

ILLOTHER, OTHERWISE is associated with a label array which includes a label with an asterisk subscript.

Error: OTHERWISE or a label with an asterisk subscript can be associated with a label array but not both.

User Action: Remove the OTHERWISE from the GOTO (or GO TO) statement.

ILLREFOBJ, A refer object *entity* is not a member of the same structure.

Error: The refer object reference must reference a refer object that is a previous member of the structure containing the REFER option.

User Action: If possible, reposition the REFER option so that it follows the refer object in the same structure.

ILLREFOPTN, The variable *entity* contains a REFER option, but is not a member of a BASED structure.

Error: The REFER option can be applied only to members of BASED structures. The program contains a REFER option that has been used with a storage class other than BASED.

User Action: Change the storage class to BASED.

ILLSUBSCRIP, Illegal label array subscript.

Error: If the label array includes a target label with an asterisk subscript, the label array subscript must be a scalar variable, an aggregate member, or a function call whose parameter is a scalar variable or aggregate member.

User Action: Remove the condition causing the error.

ILLTYPEREF, The variable *entity* is declared with the TYPE attribute and references a structure *entity* that contains a REFER option.

Error: The REFER option conflicts with the TYPE attribute because the REFER option dynamically remaps storage for a structure.

User Action: Rename the variable or remove either the TYPE attribute or the REFER option from the declaration.

IMPLBLTIN, *Entity* 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.

INCSYN, Invalid syntax in %INCLUDE statement. The correct syntax is "%INCLUDE 'file-spec';", "%INCLUDE text-module-name;" or "%INCLUDE 'text-library-name(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, apostrophes must not be used and the module name must not contain any punctuation marks.

INITCVT, One of the initial values specified for *entity* 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 invalid data type, and correct it.

INITVALUE, The CDD description for structure item *entity* contains the Initial Value attribute. Initial Value is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Initial Value attribute.

User Action: None.

INNOTAREA, The reference in the IN option of an ALLOCATE or FREE statement must be to an AREA.

Error: AREAs are the only variable type that can have other variables allocated inside them.

User Action: Correct the IN option to refer to an area.

INVALCVT, A value used in this statement cannot be converted to the data type required by the context in which it is used.

Error: An operand in an expression is not compatible with the data type required for the expression; for example, a pointer value is used in an arithmetic operation such as addition.

User Action: Verify the data types of the variables in the expression, and correct the statement.

INVATTRSYN, Invalid syntax in attribute list or duplicate attribute near *entity*.

Error: An attribute list contains an erroneous symbol, for example, a comma or parenthesis, where the compiler does not expect it. This error also occurs if the same attribute is specified more than once in the declaration of the attribute.

User Action: Verify the syntax of the attributes specified in the declaration. Check that a precision or length extent, if present, is enclosed in parentheses, that commas are present in factored declarations or in structure declarations, and so on.

INVBIFPREC, The precision specified for the result of this built-in function exceeds the implementation's maximum allowed for the resulting data type.

Error: An argument list for a built-in function contains a precision argument that is invalid.

User Action: Correct the precision argument in the built-in function reference that caused the error.

INVDESCLVL, A descriptor is an apparent structure member, but does not immediately follow a descriptor with a level number.

Error: A structure in a parameter descriptor is declared incorrectly, or an extraneous integer precedes an attribute in a parameter descriptor.

User Action: If the parameter 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 parameter is not a member of a structure, check the syntax of the descriptor and remove the number preceding the parameter.

INVFACTLVL, A factored level cannot be applied to *entity*.

Error: A declaration containing a structure is factored with an attribute that conflicts with a variable declared in the structure.

User Action: Verify the syntax of the declaration and determine the attributes that are in error. If necessary, revise the structure declaration so that attributes are not factored.

INVLABL, *Entity* is a label prefix previously declared.

Error: More than one statement has the same label. The compiler cannot resolve references to the label.

User Action: Examine the source program, and change the label or labels that are duplicates. Verify that all references to the labels are also corrected.

INVOPTN, *Entity* is an invalid option for this statement or it is incorrectly specified.

Error: A statement contains an invalid option.

User Action: Check whether keyword options are misspelled or if any OPTIONS options are specified without the OPTIONS keyword. Correct the specification.

INVPARM, *Entity* is a parameter but has been declared with a storage class or as a label.

Error: The declaration of a variable that is in the procedure's parameter list contains the BASED, CONTROLLED, DEFINED, AUTOMATIC, or STATIC attribute; or the name of a parameter is specified as a label.

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.

INVSTAREXT, An asterisk is not a valid subscript or argument.

Error: You cannot use an asterisk (*) as a subscript or an argument.

User Action: Remove or replace the asterisk.

INVSTARUSE, *Entity* is declared with an asterisk as its 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 with a constant or a valid variable declaration.

INVSUBLABL, *Entity* 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.

INVZEROLVL, Structure level numbers must be greater than zero.

Error: The level number for a structure member must not be zero.

User Action: Correct the level number so that it is in the range 1 through 32767.

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, *Entity* has been declared with a variable or incorrect iteration factor. An iteration factor of 1 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.

LARGEDST, Unable to write debugger information. Submit an SPR with a problem description.

Fatal: An internal compiler error occurred during an attempt to write debugging information.

User Action: Submit an SPR with the program that caused this error. The program can be recompiled successfully if the /DEBUG option is removed from the command line.

LEAVE, *Entity* is not a LABEL constant of a DO statement in this block.

Error: The LEAVE statement cannot transfer control out of the block that contains it. Furthermore, the LEAVE statement can only transfer control forward in the program. Therefore, the LABEL CONSTANT must appear forward in the program text.

User Action: Check that the LABEL CONSTANT has been spelled correctly. Also check that the LABEL CONSTANT is forward in the same block as the LEAVE statement.

LIKEHASMEM, Only structures without members can be declared with the LIKE attribute.

Error: The structure declaration containing the LIKE attribute already contains other members.

User Action: Remove the members of the structure or substructure containing the LIKE attribute.

LOCNEED, *Entity* 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.

MINDIGITS, The CDD description for structure item *entity* specifies precision less than allowed for the data type. Minimum precision has been supplied.

Informational: Some Common Data Dictionary data types specify a number of digits that is incompatible with PL/I data types. The PL/I compiler has expanded the data type to conform to a PL/I data type.

User Action: None.

MINOCCURS, The CDD description for structure item *entity* contains the Minimum Occurs attribute. Minimum Occurs is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Minimum Occurs attribute.

User Action: None.

MINUSCAL, *Entity* has been declared with a negative size. A size of 1 has been supplied.

Warning: A negative number is specified for a variable's extent.

User Action: Check the declaration of the character-string or bit-string variable, and change the length to a positive value.

MISDEFINE, The DEFINED variable *entity* does not match its specified base reference as required by the rules for defining.

Warning: The data type attributes of a variable with the DEFINED storage class do not match the attributes of the variable on which it is defined.

User Action: If the mismatch is acceptable, you need do nothing. Otherwise, correct the declaration so that it specifies the correct attributes.

MIXEDSTAR, *Entity* has been referenced with mixed asterisk and constant bounds. If any bound is an asterisk, all bounds must be asterisks.

Error: The declaration of an array's dimensions contains both constants and asterisks for bounds; or, when initializing an array, both constants and asterisks were used for subscripts.

User Action: Determine whether the array's bounds are known within the current procedure or the array is a parameter. If the bounds are known, specify them in the array declaration. If the array is a parameter whose bounds are not known, specify all extents as asterisks. When using asterisks to specify the entire array for initialization, all subscripts must be asterisks.

MOREERRORS, There are more errors in this statement.

Error: The compiler cannot continue interpreting the statement.

User Action: Correct the previous errors.

MULTLABL, Multiple labels on a statement are not permitted. To achieve the effect of multiple labels on any statement except a PROCEDURE, ENTRY, or FORMAT statement, write L1:; L2:; statement.

Error: A statement contains more than one label.

User Action: Delete the extraneous label, or follow it with a null statement.

NEEDSCALAR, The array or structure value *entity* has been used in a context that requires a scalar value.

Error: An aggregate was used when a scalar value was required.

User Action: Replace the reference to the aggregate with a reference to a scalar value.

NEEDVECTOR, A scalar value is being passed to a vector parameter.

Error: A scalar value cannot be passed as an actual parameter to a routine whose formal parameter is declared as a vector.

User Action: Change the actual and formal parameters to either both vectors or both scalars.

NEGSIZE, A computed string length or aggregate size is negative.

Error: The compiler calculated the length of a character string or bit string or an aggregate specified with nonconstant extents, and the result is negative.

User Action: Determine how the extent came to be in error and respecify it.

NESTDEPTH, The nesting of DO, SELECT, PROCEDURE, and BEGIN statements exceeds the implementation's limit of 64.

Fatal: There are too many nested blocks. Nesting of DO, PROCEDURE, and BEGIN statements must not exceed 64. Nesting of SELECT statements must not exceed 31.

User Action: Simplify the nesting of blocks in the program.

NESTEDPSV, This statement contains a nested pseudovariable reference to *entity*.

Error: One of the built-in functions that may be used on the left side of an assignment contains a reference to another pseudovariable. Nested use of pseudovariables as in SUBSTR(UNSPEC(x),3,1) = '0'b; is invalid.

User Action: Correct the expression so that it does not contain a nested pseudovariable reference.

NOCNDVAL, A value is required with the *entity* condition.

Error: The ENDFILE, ENDPAGE, KEY, UNDEFINEDFILE, or VAXCONDITION condition name is specified without a value.

User Action: Specify a value for the condition.

NODATATYP, *Entity* is declared without a data type. The default data type FIXED BINARY has been supplied.

Warning: The indicated name has been declared, but is declared without a data type attribute. The compiler provides the default data type of FIXED BINARY.

User Action: If you want the variable to have the default data type, you need do nothing. Otherwise, correct the declaration of the variable so that it has a data type attribute.

NODFLOAT, The CDD description for structure item *entity* specifies the D_Floating data type. The data cannot be represented when compiled with /G_FLOAT.

Warning: The wrong PLI command qualifier was used to compile the program.

User Action: Ignore the warning message or recompile the program using the /NOG_FLOAT qualifier.

NODIM, *Entity* is an entry or file constant and cannot be declared with a dimension.

Error: The ENTRY or FILE attribute was specified for an array, but the VARIABLE attribute is not specified.

User Action: Specify the VARIABLE attribute to create an array of file or entry variables.

NOGFLOAT, The CDD description for structure item *entity* specifies the G_Floating data type. The data cannot be represented when compiled with /NOG_FLOAT.

Warning: The wrong PLI command qualifier was used to compile the program.

User Action: Ignore the warning message or recompile the program using the /G_FLOAT qualifier.

NOHFLOAT, The CDD description for structure item *entity* specifies the H_Floating data type. The data cannot be represented when compiled with /NOG_FLOAT.

Warning: The wrong PLI command qualifier was used to compile the program.

User Action: Ignore the warning message or recompile the program using the /G_FLOAT qualifier.

NOINALL, The IN option is not allowed with CONTROLLED variables.

Error: CONTROLLED variables cannot be allocated in an AREA.

User Action: Either remove the IN option to allow the variable to be allocated normally, or use a BASED variable instead of a CONTROLLED variable if you really want the allocation to be inside the specified area.

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.

NOLABSUB, This statement contains a reference to an undefined subscripted label array element *entity*(*entity*).

Error: The compiler cannot resolve a reference to the indicated subscripted program label.

User Action: Verify that the label is correctly specified, and, if it is an element of an array of label constants, that the label is properly subscripted in the source program.

NOLOCNEED, *Entity* 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, *Entity* 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, *Entity* 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.

NONCONUNION, *Entity* is a member of a UNION but does not have constant size.

Error: A UNION is a variation of a structure in which all immediate members occupy the same storage. Consequently, all members of a UNION must have a constant size.

User Action: Remove the VARYING attribute from the structure declaration.

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.

NOSETALL, *Entity* is a CONTROLLED variable; this ALLOCATE statement cannot have the SET option.

Error: The SET option sets a pointer variable to the memory location of storage acquired for a BASED variable. A CONTROLLED variable cannot be used in a pointer-qualified reference.

User Action: Remove the SET option or change the variable declaration from CONTROLLED to BASED.

NOTARITH, Implicit conversion. A nonarithmetic expression, *entity*, has been used in a context requiring an arithmetic value.

Warning: A bit- 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 use 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 an LBOUND, HBOUND, DIM, PROD, or SUM 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 *entity* is not a BASED or CONTROLLED 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.

NOTBIT, Implicit conversion. A nonbit expression, *entity*, has been used in a context requiring a bit-string value.

Warning: An arithmetic or character-string expression was used in a context where a bit string is required. The PL/I compiler has converted the expression to a bit string. 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 a bit string, use the BIT built-in function to convert the character-string or arithmetic expression to a bit string. You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

NOTCHAR, Implicit conversion. A noncharacter expression, *entity*, has been used in a context requiring a character-string value.

Warning: An arithmetic or bit-string expression was used in a context where a character string is required. The PL/I compiler has converted the expression to a character string. 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 a character string, use the CHARACTER built-in function to convert the arithmetic or bit-string expression. You can also suppress the message by compiling the program with the /NOWARNINGS qualifier.

NOTCONDVAL, The name given in the CONDITION condition must be a declared CONDITION.

Error: Only names explicitly declared with the CONDITION attribute in a DECLARE statement can be used with the CONDITION condition.

User Action: Declare the condition name explicitly.

NOTCONNECT, This statement contains an invalid reference to an unconnected array.

Error: A member of a structure that has the dimension attribute is referenced in an invalid context. Because the storage of such an array is not contiguous, the array cannot be referenced in any of the contexts listed in the message.

User Action: If possible, change the declaration of the structure so that the array that caused the error becomes a connected array. Otherwise, do not reference the array in the context that caused the error.

NOTDIM, Invalid dimension specified in HBOUND, LBOUND, DIM, PROD, or SUM.

Error: The value specified for the dimension argument of the built-in function that caused the error is invalid for the array. For example, this error occurs if a value of 10 is specified for an array that has only 5 dimensions.

User Action: Determine the number of dimensions in the array and correct the argument.

NOTDIMVAL, The second argument of the LBOUND, HBOUND, or DIM built-in function must be an integer constant.

Error: The dimension argument for one of the functions listed is specified using a nonconstant expression.

User Action: Specify an integer constant for the argument.

NOTEXEC, A FORMAT, ENTRY, PROCEDURE, END, or DECLARE statement appears in a context that requires an executable single statement, DO-group, SELECT-group, or begin block.

Error: ON statements and THEN, ELSE, WHEN, and OTHERWISE clauses require that the target action be an executable statement, a DO-group, SELECT-group, or a begin block.

User Action: Move the statement in error. If appropriate, place the statement in a begin block.

NOTFILEVAL, The name given in an I/O condition must be the name of a file value.

Error: The UNDEFINEDFILE, ENDFILE, or KEY condition name is specified with a value that is not a file reference.

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, *entity*, 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.

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.

NOTPARAM, Illegal use of the PRESENT function. Its argument must be a parameter.

Error: The argument of the PRESENT built-in function is invalid. It must be a parameter. It cannot be a variable, expression, or constant.

User Action: Change the argument of the PRESENT built-in function to a parameter name.

NOTPLIDATA, BIT_FIELD and BYTE_FIELD data can be referenced only in contexts that do not require a data type interpretation.

Error: A reference to data declared as BIT_FIELD or BYTE_FIELD was made in an illegal context.

User Action: BIT_FIELD and BYTE_FIELD data can be referenced only in the ADDR, INT, POSINT, BYTESIZE, SIZE, and UNSPEC built-in functions, or as an ANY parameter. If a data type interpretation is required, define a BASED or DEFINED variable that overlays the storage of the variable.

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.

NULLARG, *Entity* has been referenced with too many empty argument lists.

Error: A procedure call or function reference specified too many empty argument lists, for example, F()()(A,B).

User Action: Determine the correct number of empty argument lists, and correct the reference.

NULLPTR, The pointer or offset in a reference to *entity* is NULL.

Error: A pointer or offset reference in a locator-qualified reference has a null value. The reference cannot be resolved.

User Action: Verify that the correct pointer or offset variable is referenced in the statement and that it was properly given a value, either with a SET option of ALLOCATE or by assignment with an ADDR, POINTER, or OFFSET built-in function.

OBJNOTDCL, The refer object *entity* is not declared.

Error: A refer object refers to a variable that has not been declared, or the variable follows the refer object reference in the source program.

User Action: Verify that the refer object reference has been declared, or reposition the REFER option so that it follows the declaration of the refer object reference.

OBJNOTMEM, The refer object *entity* is not a structure member.

Error: The refer object must be a member of the structure declared with the REFER option.

User Action: Declare the refer object within the structure that contains the REFER option.

OFFBASINV, A base area was specified for the OFFSET parameter or descriptor *entity*.

Error: An offset parameter or argument specifies a base area.

User Action: Remove the area specification from the parameter's declaration or from the parameter descriptor for the argument.

OFFSETNOBASE, In a conversion between pointer and offset data, the offset data does not have an associated base area.

Error: The compiler cannot resolve the location of the specified offset without the specification of the area in which the offset is based.

User Action: Correct the declaration of the offset so that it specifies a base area.

ONUNIT, An IF, ON, RETURN, DO, or SELECT statement cannot be used as the first statement of an ON-unit.

Error: An ON-unit contains an invalid action statement.

User Action: Correct the ON-unit so that the first statement is not an IF, ON, RETURN, DO, or SELECT statement. To execute more than one statement in an ON-unit, use a begin block.

ONUNITLABL, A label prefix cannot appear on a statement used as an ON-unit.

Error: An ON-unit action statement contains a label.

User Action: Remove the label from the statement.

PICNOTALL, The CDD description for the structure item *entity* specifies a PICTURE FOR PLI in conjunction with a data type that is not a numeric string. PICTURE FOR PLI is being ignored.

Informational: The picture specification in the Common Data Dictionary description does not conform to PL/I picture specifications.

User Action: Change the data type to conform to PL/I pictures, remove PICTURE FOR PLI, or ignore the warning message.

PICNOTALW, The PICTURE variable *entity* has been used in a context requiring a FIXED or FLOAT value.

Error: A variable having the PICTURE data type has been used in a context requiring a FIXED or FLOAT variable.

User Action: Change the data type of the variable or use one of the built-in conversion functions to convert the variable to either FIXED or FLOAT.

PLACEEOL, Placeholder not terminated before end of line.

Error: Placeholders cannot cross line boundaries.

User Action: Modify the source code so that each placeholder begins and ends on the same line.

PLACELONG, Placeholder too long.

Error: Placeholders are limited to 256 characters.

User Action: Reduce the length of the placeholder. The 256-character limit includes the outer brackets.

PLACENODESIGN, Placeholders invalid without /DESIGN=PLACEHOLDERS.

Error: Placeholders are not allowed unless /DESIGN=PLACEHOLDERS was specified.

User Action: Specify the /DESIGN=PLACEHOLDERS qualifier or remove the placeholders from the source file.

PLACENODOT, Invalid pseudocode list placeholder detected.

Error: Pseudocode placeholders are not to have the list option.

User Action: Remove the ellipsis (. . .) following the placeholder.

PLACENOOBJ, Placeholders detected - no object code generated.

Warning: The compiler does not produce object code when the source contains placeholders.

User Action: None.

PLACESYNTAX, Invalid placeholder syntax detected.

Error: Nesting of pseudocode placeholders is not allowed, or a mismatch in outer brackets was detected.

User Action: Correct the invalid placeholder.

PLACEUNMAT, Unmatched placeholder delimiter.

Error: The placeholder delimiters do not match correctly.

User Action: Correct the source code so the delimiters match.

PPARMDCL, The name *entity* has previously been declared as a preprocessor procedure parameter. It cannot be declared as a preprocessor label in this procedure.

Error: The same name was used for both a preprocessor procedure parameter and a label.

User Action: Determine which name is correct; rename either the label or the parameter.

PPBADPAREN, This preprocessor STATEMENT procedure invocation contains unbalanced parentheses.

Error: The invocation of a preprocessor procedure as a pseudo-statement contains unbalanced parentheses.

User Action: Examine the invocation and add or remove opening or closing parentheses as necessary.

PPBADRET, %RETURNS cannot be used except in a preprocessor procedure.

Error: A %RETURN statement appears in a context other than in a preprocessor procedure.

User Action: Remove the %RETURN statement or reposition it so that it is contained within a preprocessor procedure.

PPBIFARG, The *entity* preprocessor built-in function has been referenced with the wrong number of arguments.

Error: An argument list for a preprocessor built-in function contains an invalid number of arguments.

User Action: Correct the number of arguments to the built-in function.

PPBIGEXPR, An expression or reference contained within this preprocessor usage is excessively complex.

Error: The compiler cannot follow the flow of the preprocessing due to an expression or reference that is too complex.

User Action: Simplify the preprocessor statement.

PPCONVERR, Evaluation of a preprocessor expression caused a CONVERSION error.

Error: The indicated expression does not have a valid preprocessor computational data type. The operands of a preprocessor expression can consist only of unsubscripted references to preprocessor variables, decimal integer constants, bit-string constants, character-string constants, and references to preprocessor built-in functions. For arithmetic operations, only decimal integer arithmetic of precision (10,0) is performed.

User Action: Examine the expression and correct the error.

PPDCLREP, The name *entity* has previously been declared as a preprocessor variable. It cannot be declared as a %REPLACE identifier.

Error: Preprocessor variables cannot have the same name.

User Action: Rename one of the variables so that there are no duplicate names.

PPDEFPARAM, A preprocessor procedure references the undeclared preprocessor parameter *entity*. It is being declared as a FIXED preprocessor parameter.

Warning: A preprocessor parameter that was not declared has been referenced; the compiler gives the attribute FIXED by default.

User Action: Check that the reference is correctly spelled; if it is not, correct the spelling reference. If the parameter was not declared, declare it with the appropriate attribute.

PPDEFTYPE, A %*entity* statement references the undeclared preprocessor name *entity*. It is being declared as a FIXED preprocessor variable.

Warning: A name that is not declared or that is not a label has been referenced; the compiler supplies the attribute FIXED by default.

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 attribute for its use: BIT, FIXED, or CHARACTER.

PPDUPARAM, *Entity* is a duplicate preprocessor procedure parameter.

Error: The same parameter name is used more than once in a preprocessor procedure.

User Action: Correct the preprocessor procedure statements so that there is only one reference to each parameter.

PPDUPDCL, The name *entity* has already been declared as a preprocessor variable.

Error: The same identifier is used in more than one preprocessor declaration at the same level.

User Action: Determine which preprocessor declaration of the variable specifies the incorrect attributes, if they are different, and correct the declarations.

PPELSENOT, A %ELSE does not properly correspond with a %THEN.

Error: A %ELSE clause was used without a corresponding %THEN clause.

User Action: Verify that the %ELSE clause is not extraneous. If it is not, add a %THEN clause to the program, and, if no action is desired, follow it by a preprocessor null statement.

PPENDNODO, A %END does not properly correspond with a %DO.

Error: A group contains a %END statement without a preceding %DO statement.

User Action: Verify the need for a %END statement. If it is extraneous, remove it. If it terminates a preprocessor %DO-group, include the %DO statement.

PPEXPRSYN, Invalid expression in a %*entity* statement. *Entity* was found where *entity* was expected.

Error: The operands of a preprocessor expression can consist only of unsubscripted references to preprocessor variables, decimal integer constants, bit-string constants, character-string constants, and references to preprocessor built-in functions. Furthermore, only decimal integer arithmetic of precision (10,0) is performed.

User Action: Examine the preprocessor expression, and correct the error.

PPFIXOVER, Evaluation of a preprocessor expression caused the FIXEDOVERFLOW condition.

Error: The evaluation of a preprocessor expression resulted in a number that has exceeded the maximum precision. For BIT and BINARY numbers, the maximum precision is 31; for DECIMAL numbers, the maximum precision is 10.

User Action: Simplify the expression.

PPGOTOBACK, A %GOTO statement references a previously declared label. Backwards %GOTO and %GOTO out of preprocessor procedures is not permitted.

Error: The PL/I for OpenVMS VAX embedded preprocessor permits only forward scanning for labels corresponding to a %GOTO statement.

User Action: Verify that the label is properly specified or rename the label so that the %GOTO is forward.

PPGOTOEND, The target of a %GOTO statement, *entity*, was not found.

Error: The compiler cannot locate the specified target label for a %GOTO statement, or the target label is missing.

User Action: Verify that the target label exists. If no matching label exists, include one. Make certain that corresponding labels are spelled the same.

PPGOTOSYN, Invalid syntax in a %GOTO statement. The target of a %GOTO statement must be an unsubscripted label identifier.

Error: A %GOTO target is incorrectly specified or it is not an unsubscripted label.

User Action: Correct the syntax of the %GOTO statement, and verify that it is an unsubscripted label.

PPINVLABL, *Entity* is a preprocessor label prefix previously declared.

Error: The same preprocessor label has been declared for more than one statement, resulting in ambiguous label references.

User Action: Rename the preprocessor label.

PPINVRADIX, An invalid radix was specified in the encode or decode preprocessor function.

Error: The radix specified in the encode or decode preprocessor function was invalid. It must lie in the range 2 through 16.

User Action: Correct the radix value.

PPINVSTRING, An invalid string was specified in the decode preprocessor function.

Error: The string specified in the decode preprocessor function contains invalid characters. They must lie in the range 0 through the radix – 1.

User Action: Correct the string.

PPLABDCL, The name *entity* has previously been declared as a preprocessor label. It cannot be declared as a preprocessor variable.

Error: The same name has been used as both a preprocessor label and a variable name. Therefore, the compiler cannot follow program flow.

User Action: Rename either the label or the variable so that there are no duplicate names or declarations.

PPLABREP, The name *entity* has previously been declared as a preprocessor label. It may not be declared as a %REPLACE identifier.

Error: Preprocessor labels and identifiers cannot have the same name.

User Action: Rename either the label or the variable so that there are no duplicate names.

PPLABSYN, Invalid syntax in preprocessor label. A label must be of the form %identifier:, and cannot follow %THEN or %ELSE.

Error: A preprocessor label has been incorrectly specified. The correct syntax is %label: and cannot follow %THEN or %ELSE.

User Action: Examine the preprocessor label, and make certain that the percent sign and colon are positioned properly. It is not necessary to include another percent sign on that line.

PPMULTLABL, Multiple labels on a preprocessor statement are not permitted. To achieve the effect of multiple labels, write %L1:; %L2: statement.

Error: No statement, including preprocessor statements may have more than one label.

User Action: A statement can, however, be preceded by any number of labeled null statements. To achieve the effect of multiple preprocessor labels, write %L1:; %L2: in your source program.

PPNEST, Nested preprocessor procedures are not allowed.

Error: A %PROCEDURE statement cannot be used within a preprocessor procedure.

User Action: Restructure the preprocessor procedures so that they are not nested.

PPNOFILE, Cannot access source file for preprocessor scanning.

Fatal: The compiler cannot access an include file; it cannot be found on the disk.

User Action: Verify that the file is accessible and recompile.

PPNORET, Preprocessor procedure exited without execution of %RETURN.

Error: The %RETURN statement is required in preprocessor procedures. Values are passed back to the invoking source program by means of the %RETURN statement.

User Action: Include a %RETURN statement in your program.

PPNOTHEN, %IF is not terminated with %THEN.

Error: A preprocessor %IF-group is not followed by a %THEN statement.

User Action: Either include a %THEN statement or delete the %IF statement.

PPNOTSTMT, The clause following %THEN or %ELSE is not a preprocessor statement.

Error: Clauses following the %THEN or %ELSE statements require a preprocessor statement. Otherwise the statement cannot be executed. Valid actions of the %THEN and %ELSE clauses are preprocessor statements.

User Action: Verify that the statement is a preprocessor statement preceded by a percent symbol. It must not be a %END statement.

PPRECURS, The preprocessor procedure *entity* has been invoked recursively 999 times. No further recursion will be performed.

Error: A preprocessor procedure has called itself 999 times.

User Action: Correct or simplify the program.

PPREPDCL, The name *entity* has previously been declared as a %REPLACE identifier. It cannot be declared as a preprocessor variable.

Error: Preprocessor statements cannot have duplicate identifiers, and %REPLACE is a preprocessor statement.

User Action: Rename one of the identifiers so that there are no duplicate declarations.

PPREPLAB, The name *entity* has previously been declared as a %REPLACE identifier. It may not be declared as a preprocessor label.

Error: Preprocessor labels cannot have the same name as identifiers.

User Action: Rename either the identifier or the label so that there are no duplicate names.

PPSTMTSYN, Invalid syntax in a %*entity* statement. *Entity* was found where *entity* was expected.

Error: A preprocessor statement was improperly specified. The format of a preprocessor statement is as follows:

```
%[label:]STATEMENT;
```

User Action: Correct the syntax.

PPSTRINGSIZE, The length of a preprocessor character string exceeds the implementation limit of 32500.

Error: A preprocessor expression yields a string larger than the implementation limit.

User Action: Attempt to reduce the size of the string or segment it among several variables.

PPSTRRANGE, Evaluation of a preprocessor expression caused the STRINGRANGE condition.

Error: The STRINGRANGE condition (string index out of bounds) was detected during evaluation of a preprocessor expression.

User Action: Correct the preprocessor source code to have only valid string and substring references.

PPSUBSTR2, The second operand of a preprocessor SUBSTR is out of range.

Error: The second operand in a reference to a SUBSTR preprocessor built-in function is beyond the range of the string. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

PPSUBSTR3, The third operand of a preprocessor SUBSTR is out of range.

Error: The third operand in a reference to a SUBSTR preprocessor built-in function is beyond the range of the string. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

PPTEXTRA, A %*entity* statement contains extraneous text where ";" was expected.

Warning: Nonpreprocessor text follows a preprocessor statement.

User Action: Locate the beginning of the extraneous text and check for unbalanced or missing comment delimiters (/* or */).

PPTHENNOIF, A %THEN does not properly correspond with a %IF.

Error: A %THEN statement must have a corresponding %IF statement. The format of the %IF statement is as follows:

```
%IF test-expression %THEN action [%ELSE action];
```

User Action: Verify that the syntax is correct.

PPTOOBIG, Preprocessor text expansion exceeds the implementation limit for number of characters per line.

Error: A source input record has more than 255 characters, which is the capacity of the input buffer.

User Action: Ensure that the RESCAN option has been used properly and is not needlessly replacing characters. Otherwise, reduce the size of the text and recompile.

PPTOODEEP, Combined %DO and %IF nesting has exceeded the implementation limit for maximum depth of nesting.

Error: There are too many nested %DO and %IF groups.

User Action: Simplify the nesting of %DO and %IF groups in the program.

PPTOOFEW, Insufficient %END statements. An additional %END statement has been supplied.

Warning: The compiler inserted a %END statement in the program so that the number of %ENDs is correct.

User Action: Match the %END statements with %DO statements to ensure that each preprocessor group is properly terminated.

PPTOOMANY, Preprocessor replacements have been applied to this statement 999 times. No further replacements will be performed.

Warning: Replacement has exceeded the implementation limit of 999.

User Action: Ensure that the RESCAN option has been used properly and is not needlessly replacing characters. Otherwise, reduce the number of replacements.

PPUNRFUNC, *Entity* is not a preprocessor built-in function known to this implementation.

Error: A function does not contain any recognizable PL/I preprocessor statements or keywords, or a statement contains extraneous names.

User Action: Examine the function in error and correct it.

PPVARDCL, The name *entity* has previously been declared as a preprocessor variable. It may not be declared as a preprocessor label.

Error: The same name has been used as both a preprocessor label and a variable name. Therefore, the compiler cannot follow program flow.

User Action: Rename either the label or the variable so that there are no duplicate names or declarations.

PPZERODIV, Evaluation of a preprocessor expression caused the ZERODIVIDE condition.

Error: The divisor in a division operation has the value of zero. The value resulting from such an operation is undefined.

User Action: Recompile the program with a valid expression.

PROLOGVAL, The value of the AUTOMATIC or DEFINED variable *entity* is used in an extent expression or initial-value expression in the declaration of an AUTOMATIC or DEFINED variable in the same block.

Warning: Dependency on the value of an AUTOMATIC or DEFINED variable that is declared in the same block is not allowed. When the compiler generates code for the variable, the referenced automatic or defined variable may not have been allocated or initialized.

User Action: Correct the extent or initial-value expression so that it specifies a constant value, a static variable, or parameter, or so that it references a variable that is defined in a containing block.

PTQTYPEDEF, The variable *entity* TYPE reference is a pointer-qualified variable which is not allowed.

Error: A pointer-qualified TYPE reference implies that the typed variable can inherit, from the type definition, the BASED attribute which is not true.

User Action: Declare the variable with explicit attributes.

PUTGETUNION, The union *entity* occurs as a source or target in a GET or PUT statement.

Warning: The UNION attribute must be associated with a level number in a structure declaration, but structure declarations containing the UNION attribute cannot be used in I/O statements.

User Action: Remove the UNION attribute.

REFERENCE, The CDD description for structure item *entity* contains the REFERENCE attribute. REFERENCE is being ignored by PL/I.

Informational: The REFERENCE attribute is not supported by PL/I.

User Action: None.

REFERENCEBIF, Invalid use of the REFERENCE built-in function.

Error: The REFERENCE built-in function can be used only to override the passing mechanism of an argument. This means it must be used in an argument list, and its parameter must be an argument.

User Action: Remove the reference to the built-in function from the statement.

REPLSYN, Invalid syntax in a %REPLACE statement.

Error: A %REPLACE statement is specified incorrectly.

User Action: Correct the statement.

REQINIT, An INITIAL attribute must be specified for *entity*.

Warning: The indicated name is not specified with the INITIAL attribute. This error occurs for names declared with the READONLY or VALUE attributes. Because 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.

RESIGNOTON, RESIGNAL built-in function has been referenced from outside an ON unit.

Error: The RESIGNAL built-in function has been referenced in a block that is not the first level block of an ON-unit.

User Action: The RESIGNAL built-in function cannot be referenced from outside an ON-unit or in a nested block of an ON-unit. Control must be passed back down to the immediate block of the ON-unit through a nonlocal GOTO before RESIGNAL is called.

RETANY, A returns descriptor must not specify ANY as its data type for *entity*. FIXED BINARY(31) has been forced.

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 for *entity*.

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 for *entity*.

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 *entity* to the function type *entity*.

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 will be

issued once for each occurrence of a RETURN statement that requires an implied conversion.

User Action: If the conversion is desirable, 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.

RETVALXX, The value *entity* in a RETURN statement is not valid for conversion to the *entity* function type of one of the entry points of this multi-entry-point procedure.

Warning: This warning message is given only for a RETURN statement in a procedure that has multiple entry points. It is issued when a RETURN statement specifies a return value that is incompatible with the data type specified in the corresponding returns descriptor of one or more of the entries. The message is issued once for each incompatible return value.

User Action: Verify that the program's flow of control is such that the invalid conversion will not take place. To avoid this message, correct the return data type in one or more RETURN statements.

RIGHTJUST, The CDD description for structure item *entity* contains the Right Justified attribute. Right Justified is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Right Justified attribute.

User Action: None.

ROUNDARG2, The first argument of the ROUND built-in function must be fixed decimal or pictured or fixed binary and must have a positive scale factor. The second argument must be an integer constant in the range 0 through 31.

Error: A reference to the ROUND built-in function specifies an invalid argument list.

User Action: Correct the argument in the reference to the ROUND built-in function.

SBSTYPEDEF, The variable *entity* TYPE reference is a subscripted variable which is not allowed.

Error: When an array is declared, the attributes are associated with the array and not with the array member.

User Action: Omit the subscript from the TYPE reference.

SCAERROR, Internal compiler error during SCA processing. Please submit an SPR.

Fatal: An internal compiler error during SCA processing.

User Action: Submit an SPR. Recompile the program without the /ANA qualifier.

SCALEIGNOR, *Entity* has been declared FLOAT with a scale factor. The scale factor will be ignored.

Warning: A floating-point variable has been declared with a scale factor.

User Action: Correct the declaration of the floating-point variable so that it does not specify a scale factor.

SETREQ, Because the variable *entity* was not declared with a base pointer, this ALLOCATE statement requires a SET option.

Error: The variable referenced in the ALLOCATE statement has the BASED attribute but is not declared with an explicit pointer reference.

User Action: Specify the SET option on the ALLOCATE statement, or correct the BASED variable's declaration so that it specifies a pointer variable.

SIZE, Invalid type of argument for the SIZE or BYTESIZE built-in function.

Error: The argument of the SIZE or BYTESIZE built-in function was not a reference to a variable. For example, it might have been a constant or an expression.

User Action: Supply a valid argument.

SMALLAREA, If the AREA *entity* is to be initialized it must have an extent of at least 24 bytes.

Error: The AREA variable was declared with an extent of less than 24 bytes. AREAs of this size are too small to be initialized by the EMPTY( ) built-in function.

User Action: Change the extent specified so that the area has an extent of at least 24 bytes.

SOURCETYPE, The CDD description for structure item *entity* contains the Source Type attribute. Source Type is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Source Type attribute.

User Action: None.

STAREAINIT, *Entity* has been declared with an invalid initial value. Static AREAs can be initialized only to EMPTY( ).

Error: An invalid initial value has been specified for the AREA variable referenced in the DECLARE statement. AREAs with the STATIC attribute can be initialized only with the EMPTY( ) built-in function.

User Action: Change the initial value to EMPTY( ) or change the declaration so that the variable does not have the STATIC attribute.

STMTOOBIG, A statement exceeds the implementation's limit of constants, identifiers, operators, and punctuation symbols. Or, there are more than approximately 4500 statements in the procedure.

Fatal: The statement contains more than 2048 constants, identifiers, and punctuation symbols, including operators.

User Action: Simplify the statement.

STMTSYNKEY, Invalid syntax in an *entity* statement. *Entity* was found where the *entity* keyword was expected.

Error: A statement is missing a keyword.

User Action: Correct the syntax of the statement as indicated by the message.

STMTSYNTOK, Invalid syntax in an *entity* statement. *Entity* was found where *entity* was expected.

Error: The compiler looks for specific types of tokens within statements, attribute lists, and extents. This error occurs when one type of token appears in an inappropriate context, for example, if the ADDR built-in function occurs in an INITIAL attribute list or if the required string is missing from a PICTURE attribute.

User Action: Determine what data the compiler expects and correct the statement.

STPTOFINIT, *Entity* has been declared with an invalid initial value. Static POINTERs or OFFSETs can be initialized only to NULL().

Error: An invalid initial value has been specified for the POINTER or OFFSET variable referenced in the DECLARE statement. POINTERs and OFFSETs with the STATIC attribute can be initialized only with the NULL() built-in function.

User Action: Change the initial value to NULL(), or change the declaration so that the variable does not have the STATIC attribute.

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 limit of 32500 characters. Ensure that all string constants are delimited with ' and that any contained 's occur in pairs. Also check for unbalanced /* */.

Fatal: The compiler read more than 32500 characters following the occurrence of an open apostrophe (') or comment (/*).

User Action: Terminate the unterminated string or comment 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 *entity* does not lie in the range *entity* : *entity*.

Error: An integer constant appears in a context where the value of the integer is outside the permissible range.

User Action: Check that the constant was specified correctly and that it does not exceed the implementation limit for the context.

SUBROUT, The subroutine *entity* 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, add the RETURNS attribute to the PROCEDURE statement of the procedure so that it can be invoked by a function reference.

TAGVALUES, The CDD description for structure item *entity* contains the Tag Values attribute. Tag Values is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Tag Values attribute.

User Action: None.

TAGVARIAB, The CDD description for structure item *entity* contains the Tag variable attribute. Tag variable is being ignored by PL/I.

Informational: PL/I does not support the Common Data Dictionary Tag variable attribute.

User Action: None.

TOOFEWARG, *Entity* has been referenced with too few arguments.

Error: The number of parameters in a parameter list or parameter descriptor list exceeds the number of arguments specified in the corresponding procedure reference.

User Action: Verify the number of arguments required by the invoked procedure, and correct the argument list. If the procedure has a variable-length argument list, specify the TRUNCATE attribute on the proper formal parameters in the entry declaration.

TOOFEWSUB, *Entity* has been referenced with too few subscripts. Subscripted references must have as many subscripts as the array has dimensions, including any inherited dimensions.

Error: The number of subscripts specified in the reference to the array variable is fewer than the number of dimensions in the array.

User Action: Examine the declaration of the array to determine the number of subscripts required, determine the missing subscripts, and correct the statement. If the array is declared within a structure that is dimensioned, be sure to include the structure dimensions in the total dimensions of the array.

TOOFEWVAL, The INITIAL attribute specified for *entity* contains fewer values than are required to fully initialize the variable.

Warning: An INITIAL attribute specified for an array specifies fewer items than there are elements of the array.

User Action: Verify that the program will execute successfully without all array elements initialized. If not, correct the declaration of the array so that all elements are initialized.

TOOMANYARG, *Entity* has been referenced with too many arguments.

Error: The number of arguments in a procedure reference exceeds the number of parameters in the corresponding parameter list or parameter descriptor list.

User Action: Verify the number of arguments required by the invoked procedure, and correct the argument list. If the procedure is a non-PL/I procedure with a variable-length argument list, specify the LIST attribute on the last parameter of the entry declaration.

TOOMANYDIM, More than eight dimensions have been specified in the declaration of an array.

Error: The parenthesized list of dimensions in the declaration of the array contains more than eight items.

User Action: Correct the declaration of the array so that it has no more than eight dimensions.

TOOMANYLBARS, Too many label arrays associated with GOTO OTHERWISES and/or ASTERISK LABEL SUBSCRIPTS.

Error: The number of label arrays associated with GOTO OTHERWISES and/or ASTERISK LABEL SUBSCRIPTS is limited to 500 per compilation.

User Action: Reduce the number of label arrays associated with GOTO OTHERWISES and/or ASTERISK LABEL SUBSCRIPTS.

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, *Entity* 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 subscripts in excess, and correct the statement.

TOOMANYVAL, Excess initial values have been specified for *entity*.

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.

TOTALDIM, More than eight dimensions have been specified for the array *entity*. This may include dimensions inherited from containing structures.

Error: An array has been declared with more than eight dimensions, or an array is declared within a structure, and the sum of the array's dimensions and those of the structure exceeds a total of eight.

User Action: Correct the declaration of the array.

TYPEHASMEM, Only structures without members can be declared with the *entity* TYPE attribute.

Error: The structure declaration containing the TYPE attribute already contains other members.

User Action: Remove the members of the structure or substructure containing the TYPE attribute.

UNALIGNED, *Entity* has been declared with the UNALIGNED attribute. Only BIT or CHARACTER string variables can be declared UNALIGNED.

Error: PL/I for OpenVMS VAX does not implement the UNALIGNED attribute for types other than NONVARYING strings.

User Action: Remove the UNALIGNED attribute.

UNDCLBASE, *Entity* is undeclared and has been used in an ALLOCATE statement as the name of a BASED or CONTROLLED 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 it is, declare it with the BASED or CONTROLLED attribute.

UNDCLPARM, *Entity* 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.

User Action: Declare the parameter with the appropriate data type attributes.

UNLIKEREF, *Entity*, which has been declared with the LIKE attribute, references a variable that is not known to this block.

Error: The major or minor structure referenced by the LIKE attribute is not known to the current block. Therefore, the compiler cannot locate the referenced structure.

User Action: Declare the referenced structure outside of the procedure, if the STATIC storage class is acceptable. Otherwise, declare the referenced structure within the appropriate block.

UNRATTR, *Entity* is an unrecognizable attribute.

Error: A declaration in a descriptor contains an invalid keyword.

User Action: Check for typographical errors on the statement. Verify the syntax of the descriptor, noting particularly that commas, parentheses, and spaces appear where required.

UNRCNDNAME, *Entity* is an unrecognizable condition name.

Error: An ON, SIGNAL, or REVERT statement specifies an invalid condition name.

User Action: Check the statement for a typographical error. Verify that the condition name specified is a valid PL/I for OpenVMS VAX condition name.

UNRENV, *Entity* is an unrecognized ENVIRONMENT keyword.

Error: An ENVIRONMENT option list is incorrectly specified or contains an invalid option.

User Action: Check the list of ENVIRONMENT options for a typographical error, a missing underscore character, or an invalidly abbreviated option name. Verify that all options specified are valid ENVIRONMENT options in PL/I for OpenVMS VAX.

UNRFMT, *Entity* is an unrecognizable format item.

Error: A format list in a GET EDIT or PUT EDIT statement contains a character that is not a valid PL/I for OpenVMS VAX format item.

User Action: Verify the syntax of the format list and ensure that the format item is a valid PL/I for OpenVMS VAX format item.

UNRLOCREF, A locator-qualified reference to *entity* cannot be resolved to any declaration known to this block.

Error: A reference to a BASED variable is not valid.

User Action: Check the declaration of the variable to ensure that it is correctly specified and that the qualified reference specifies a valid pointer or offset value.

UNRPERSTMT, *Entity* is an unrecognizable preprocessor statement.

Error: A statement beginning with a percent sign (%) is not a valid PL/I for OpenVMS VAX preprocessor statement.

User Action: Check that a preprocessor keyword is spelled correctly; or, if a percent sign is specified in a character-string constant, verify that the string is properly delimited.

UNRSTMT, This is an unrecognizable statement. Starting at *entity*.

Error: A statement does not contain any identifiable PL/I statement keywords, or a statement contains extraneous tokens. For example, the statement ON . . . THEN causes this error.

User Action: Examine the statement in error and correct it.

UNRSTREF, A structure-qualified reference to *entity* 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.

UNSUPPTYPE, The CDD description for structure item *entity* specifies an unsupported data type.

Informational: The Common Data Dictionary description for a structure item has attempted to use a data type that is not supported by PL/I. The PL/I compiler has supplied the data type of BYTE_FIELD or BIT_FIELD.

User Action: Change the data type to one which is supported by PL/I, or use the PL/I built-in functions ADDR, SIZE, BYTESIZE, or UNSPEC to manipulate BYTE_FIELD and BIT_FIELD.

UNTYPEREF, *Entity* has been declared with the TYPE attribute; *entity* references a variable that is not known to *entity* in this block.

Error: The major or minor structure referenced by the TYPE attribute is not known to the current block. Therefore, the compiler cannot locate the referenced structure.

User Action: Declare the referenced structure outside of the procedure, if the STATIC storage class is acceptable. Otherwise, declare the referenced structure within the appropriate block.

UPPRGTRLOW, One of the bounds declared for *entity* 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 bound x:y, the value of x must be numerically less than the value of y.

USERDIAG, *Entity*.

Error: The PL/I for OpenVMS VAX embedded preprocessor permits you to write your own compile-time diagnostic messages. The text of the message is user-specified and responds to a user-specified compile-time condition. Error messages inhibit the production of an object file.

User Action: Determine the cause of the problem from the source text and correct the error.

VALPARAM, The PRESENT function may return an unpredictable result when its argument is a value parameter.

Warning: The argument to the PRESENT built-in function is a parameter that is passed by value. Correct results may not be returned by this function, because passing a zero will cause PRESENT to return false.

User Action: Change the passing mechanism of the parameter, or make sure that a zero is never passed as an argument to this routine.

VALSIZE, The size or precision of *entity* 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 *entity* 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.

VALUEBIF, Invalid use of the VALUE built-in function.

Error: The VALUE built-in function can be used only to override the passing mechanism of an argument. This means it must be used in an argument list and its parameter must be an argument.

User Action: Remove the reference to the built-in function from the statement.

VARFORMAT, The Common Data Dictionary record description *entity* specifies variable format.

Error: PL/I does not accept variable formats from the Common Data Dictionary.

User Action: Rewrite the Common Data Dictionary record description so that the format is fixed length.

VARYING, *Entity* has been declared with the VARYING attribute. Only CHARACTER variables can 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 *entity* is not in the range 0<=q<=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, *Entity* 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.

A.2 Run-Time Messages

This section lists the diagnostic messages produced by the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP run-time systems.

ANYCOND, PL/I ANYCONDITION condition.

Fatal: This error message is displayed when ANYCONDITION is specifically signaled and there is no ON-unit for ANYCONDITION.

User Action: Write an ON-unit to handle ANYCONDITION, or do not signal ANYCONDITION. Correct the source program.

AREA, PL/I AREA condition.

Fatal: The PL/I AREA condition was raised either by a SIGNAL AREA statement, or by the PL/I run-time library.

User Action: If the exception was raised by the RTL, see the secondary condition for more information.

AREA_ACTIVE, Operation attempted on an already active area.

Informational: The PL/I run-time library detected that an area it was about to perform an operation on is already active. If this condition occurs, it is typically because an operation is being attempted on an area from both AST level and non-AST level simultaneously, or because the area is in a shared memory environment and being modified by more than one processor at once.

User Action: Rewrite the application to avoid simultaneous modification of an area. If that is not desirable, you may want to consider handling the condition with an ON-unit. See Chapter 10 for more information.

AREA_ASSIGN, Target area in area assignment is too small.

Informational: An assignment was attempted, and the size of the target area was insufficient to hold the extent of the source area.

User Action: Correct the situation that caused the problem to occur, or add an ON-unit to process the condition.

AREA_FORMAT, Operation attempted on an incorrectly formatted area.

Informational: The PL/I run-time library detected an area to be incorrectly formatted while attempting an allocation or deallocation in the area, or while performing an area assignment from the area.

User Action: The area may be corrupt because it was never initialized with the EMPTY( ) built-in function, or because of invalid pointer references or data access.

AREA_FREE, Error detected deallocating a variable in an area.

Informational: The PL/I run-time library detected that a FREE operation in an area was not valid. For example, the variable being freed incorrectly extends beyond the end of the area, extends into a free section of the area, or is already in a free section of the area. This problem normally arises when a variable of a different size than the one originally allocated is used for the free operation, or when the variable is freed more than one time.

User Action: Correct the logic problem in the source program.

AREA_FULL, No room for variable allocation in area at address *entity*.

Informational: An allocation was attempted in an area that could not be performed because the area did not have enough contiguous space to allocate the variable requested. The address of the area is in hexadecimal.

User Action: Handle the condition. See Chapter 10 for more information.

AREA_INACTIVE, Attempt to unlock an area failed.

Informational: The PL/I run-time library detected an error while attempting to unlock the interlock bit on an area after performing an operation on the area. This error can be caused by assigning to the target area while another operation is active, since area assignment cannot be fully interlocked.

User Action: Rewrite the application to avoid assignment to an area while another operation is in progress.

AREA_SIZE, Impossible to allocate variable in area.

Informational: An allocation was attempted in an area that could not be performed because the size requested was too large to be allocated from the area, even if the area was empty.

User Action: Correct the situation that caused the problem to occur.

AUTOINIT, Error in automatic initialization.

Fatal: An automatic variable declared with the INITIAL attribute cannot be initialized. This error can be caused by a negative repetition factor in an initial list, or when too few or too many values are specified in the initialization of an array. PL/I for OpenVMS VAX also does not allow automatic initialization of automatic unconnected aggregates.

User Action: Correct the initial list.

BADATTR, File attributes conflict with request.

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.

BADRTL, Invalid file control block or uninitialized FILE VARIABLE.

Informational: A PL/I program was compiled with a field-test version of the PL/I for OpenVMS VAX compiler and linked with a later version of the run-time library.

User Action: Determine what the correct run-time library is for your version of the compiler (ask your system manager or operator, if necessary). Recompile the program with the correct run-time library.

BASECTLINIT, Error in BASED or CONTROLLED initialization.

Fatal: A BASED or CONTROLLED variable declared with the INITIAL attribute cannot be initialized. This error can be caused by a negative repetition factor in an initial list, or when too few or too many values are specified in the initialization of an array. PL/I for OpenVMS VAX also does not allow automatic initialization of BASED or CONTROLLED unconnected aggregates.

User Action: Correct the initial list.

BIFSTAPOS, Starting position for a string built-in function is out of range.

Informational: The third operand in a reference to an INDEX, SEARCH, or VERIFY built-in function is beyond the range of the string. This condition is always signaled when strings are being processed by the PL/I run-time library, but it is raised for in-line code only if the /CHECK qualifier was specified for the procedure containing the reference.

User Action: Examine the program to determine the proper value for the third parameter, and correct the program.

CNVERR, Conversion error.

Informational: An input or output value could not be converted as specified by the default conversion rules (for example, in list-directed stream I/O) or as specified by the corresponding format item (for edit-directed stream I/O).

User Action: Examine the program logic and correct it if possible. If edit-directed stream I/O is being performed, check to ensure that the input or output value is matched to the correct format item.

CONAPPSUP, APPEND and SUPERSEDE conflict.

Informational: These ENVIRONMENT options conflict and must not be both specified for the same file.

User Action: Determine whether you want to append new records to the existing file or to supersede it and write a new file. Correct the source program so that one of these options is not specified in the DECLARE or OPEN statement for the file.

CONATTR, Conflicting attributes specified while opening file.

Informational: A file was implicitly opened with an attribute that conflicts with an atttribute specified in the file's declaration. This error occurs when a file is declared with any of the following: the DIRECT and SEQUENTIAL attributes, the RECORD and STREAM attributes, or more than one of the attributes INPUT, OUTPUT, and UPDATE.

User Action: Determine the correct set of attributes for the file, and correct the file's declaration.

CONBLOKIO, BLOCK_IO conflicts with other attributes or options.

Informational: The ENVIRONMENT option list for a file contains the BLOCK_IO option and one or more of the options that conflict with BLOCK_IO.

User Action: Consult the description of the BLOCK_IO option to determine the options that conflict, and examine the file's declaration and OPEN statement. Decide whether the file is to be opened for BLOCK_IO, and correct the program.

CONDITION, PL/I CONDITION(*entity*) condition.

Informational: A user-defined condition was signaled with the SIGNAL CONDITION(user-cond) statement.

User Action: Do nothing, or add an ON-unit to handle the condition.

CONENVOPT, DECLARED option conflicts with OPEN option.

Informational: The value of an option specified in the ENVIRONMENT option list in the declaration of a file conflicts with the value specified on the OPEN statement for the file.

User Action: Determine which value is the correct value for the option, and correct either the file's declaration or the OPEN statement.

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 descriptions 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 records with fixed-length control; 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 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.

CONVERSION, PL/I CONVERSION condition.

Fatal: An invalid character was detected during the conversion of character data to another data type. (For example, the colon in '12:4' would cause conversion to be raised if this string was being converted to a FIXED value.)

User Action: Correct the input data or add an ON-unit to handle the CONVERSION condition. Note that the ONSOURCE and ONCHAR built-in functions can be used to determine the source of the error, and that the corresponding pseudovariables can be used to correct the source string.

CONVFILE, On file *entity*.

Informational: This displays the name of the file constant for which the conversion error occurred.

User Action: None.

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 be opened only for UPDATE or INPUT.

User Action: Use the RMS utility program FDL 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.

ENDFILE, PL/I ENDFILE condition on file *entity*.

Fatal: This message is displayed when a READ or GET statement attempts to access data that is beyond the end of the given file. The message is displayed only when no user-specified ON-unit exists to handle the end-of-file condition for the given file.

User Action: Provide an ON-unit for the ENDFILE condition for the input file.

ENDPAGE, PL/I ENDPAGE condition on file *entity*.

Warning: This message is displayed when a PUT statement causes the current line number to exceed the page size specified for a print file. The message is displayed only if there is no ON-unit within the file to handle the ENDPAGE condition for the given file.

User Action: If your program is displaying lines on the terminal, you may want to include this statement:

```
ON ENDPAGE(SYSPRINT);
```

This null ON-unit causes PL/I to ignore the ENDPAGE condition when many lines are being written to the terminal.

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.

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, and correct the program.

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.

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.

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.

FILENAME, File name: *entity*.

Informational: This message specifies the VMS file specification of the file to which I/O was attempted.

User Action: Examine this informational message to determine the full specification of the VMS file on which the I/O 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.

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 Chapter 10.

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.

FORMATOVFL, Too many iteration factors or remote formats.

Informational: A format list is too complex to be interpreted.

User Action: Simplify the stream I/O statement.

FXCSIZ, FIXED_CONTROL_SIZE incorrect.

Informational: The size of the variable in the FIXED_CONTROL_FROM or FIXED_CONTROL_TO option does not match the size of the file's fixed-control area.

User Action: Determine the correct size of the fixed-control area, and correct the source program or verify that the correct file is being accessed.

INCRETURN, RETURN statement is incompatible with ENTRY.

Fatal: A procedure does not have the RETURNS attribute, but an entry within that procedure specifies the RETURNS option and attempts to execute a RETURN statement.

User Action: Correct the source program by specifying the RETURNS attribute on the PROCEDURE statement or by removing it from the ENTRY statement.

INTERNAL, PL/I compiler/run-time error. Please submit an SPR.

Fatal: 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.

INVBKTSIZ, Invalid BUCKET_SIZE specified.

Informational: The value specified in the BUCKET_SIZE ENVIRONMENT option is not in the range 0 through 32. The largest number of blocks allowed in a bucket by VAX RMS is 32.

User Action: Select a bucket size that is in the correct range, and correct the source program.

INVBLKSIZ, Invalid BLOCK_SIZE specified.

Informational: The value specified in the BLOCK_SIZE ENVIRONMENT option is not in the range 20 through 65532 or is not 0.

User Action: Select a block size that is in the valid range, and correct the program.

INVDATYP, Invalid data type for record I/O.

Informational: The data type of a variable in a record I/O statement is not a computational data type; or the data type is computational but is an unconnected array or structure, an unaligned bit string, or an aggregate of unaligned bit strings.

User Action: Verify that the correct variable name was specified in the I/O statement. If the variable is an aggregate, you may have to redimension or restructure it so that the required array is connected.

INVDFNAM, Invalid DEFAULT_FILE_NAME.

Informational: The expression value specified in the DEFAULT_FILE_NAME ENVIRONMENT option is not a valid character-string expression, or it is longer than 128 characters.

User Action: Verify that the expression is correctly specified, if a variable reference is specified, that the reference is correct. Correct the source program.

INVEXTSIZ, Invalid EXTENSION_SIZE specified.

Informational: The value specified in the EXTENSION_SIZE option of ENVIRONMENT is not in the range 0 through 65535 or is not a valid integer expression.

User Action: Correct the expression.

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.

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.

INVFXCSIZ, Invalid FIXED_CONTROL_SIZE specified.

Informational: The value specified in the FIXED_CONTROL_SIZE ENVIRONMENT option is not in the range 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 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 the program.

INVMAXREC, Invalid MAXIMUM_RECORD_SIZE specified.

Informational: The value specified for the MAXIMUM_RECORD_SIZE option of ENVIRONMENT is not in the range 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 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 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 –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 the program.

INVNUMOPT, 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.

INVOWNGRP, Invalid OWNER_GROUP specified.

Informational: The value specified for the OWNER_GROUP ENVIRONMENT option is not in the range 0 through 16383, or is not a valid integer expression.

User Action: Correct the program.

INVOWNMEM, Invalid OWNER_MEMBER specified.

Informational: The value specified for the OWNER_MEMBER ENVIRONMENT option is not in the range 0 through 65535, or is not a valid integer expression.

User Action: Correct the program and recompile.

INVPROT, Invalid protection string specified.

Informational: The value specified for one of the ENVIRONMENT options GROUP_PROTECTION, OWNER_PROTECTION, SYSTEM_PROTECTION, or WORLD_PROTECTION is not a valid string expression; the string contains more than four characters; or the string contains characters other than the characters R, W, E, or D, or their lowercase equivalents.

User Action: Correct the value specified in the option, and recompile the program.

INVRADIX, An invalid radix was specified.

Fatal: The radix specified for the ENCODE or DECODE built-in function was not a value between 2 and 16.

User Action: Change the radix to a value between 2 and 16.

INVRTVPTR, Invalid RETRIEVAL_POINTERS specified.

Informational: The value specified for the ENVIRONMENT option RETRIEVAL_POINTERS is not in the range –1 through 127, or is not a valid integer expression.

User Action: Verify that the expression specified in the RETRIEVAL_POINTERS option is a valid integer expression or, if a variable reference is specified, that it refers to the appropriate variable. Correct the program.

INVSKIP, Invalid value for SKIP option specified.

Informational: The value specified in a SKIP option is zero (on an input operation) or is negative (for either an input or an output operation).

User Action: Determine the value of the SKIP option; if a variable reference is specified, verify that the variable contains the correct value.

INVSTRFMT, Invalid format item for STRING I/O.

Informational: One of the format items COL, SKIP, LINE, PAGE, or TAB was specified in a GET STRING or PUT STRING statement. These format items are not valid for these statements.

User Action: Correct the format list for the statement that caused the error.

INVSTRING, Invalid character encountered in string.

Fatal: Invalid characters were specified for the string argument of the DECODE built-in function.

User Action: Remove the invalid characters.

INVTIME, Invalid timeout value specified.

Fatal: An invalid timeout value has been specified. The number of seconds to wait must be less than 256.

User Action: Correct the timeout value.

INV_KEY, Invalid KEY data type.

Informational: The data type of a key in an indexed sequential file is not a data type known to PL/I.

User Action: Verify that the file has not been corrupted. Revert to an earlier version of the file, if possible.

IOERROR, I/O error on file *entity*.

Informational: This informational message indicates that an error occurred during an I/O operation.

User Action: Examine the accompanying messages to determine the error.

KEY, PL/I KEY condition on file *entity*.

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.

LABELRANGE, Uninitialized label subscript used.

Fatal: A label subscript is specified with a variable reference, but the variable is not initialized.

User Action: Correct the program's logic so that the variable has a valid value.

LINESIZE, Invalid LINESIZE specified.

Informational: The value specified in the LINESIZE option exceeds the implementation's limit of 32767 or the value is not a positive integer value.

User Action: Correct the LINESIZE option.

LINOVRFLO, Line number overflow.

Informational: The number of lines on a stream file page exceeds the implementation's limit of 32767.

User Action: Write an ON-unit using the VAXCONDITION or ANYCONDITION to handle the condition.

NAME, PL/I compiler/run-time error. Please submit an SPR.

Fatal: 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.

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 it specifies the SET option, or correct the REWRITE statement so that it 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.

NOSELECT, No WHEN clause selected and no OTHERWISE specified.

Fatal: It is possible to omit either the WHEN or OTHERWISE clause, but if no WHEN clause is selected, then an OTHERWISE clause must be present.

User Action: Include an OTHERWISE clause in the SELECT-group.

NOSHARE, SHARED_READ or SHARED_WRITE conflict 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.

NOTIMPL, The image being run requires a more recent version of PLIRTL.

Fatal: This message is displayed when an image containing PL/I code is moved to a system with an older version of PLIRTL that does not support a function that is required to run the program.

User Action: Upgrade the system to the necessary level to run 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.

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.

NOTPRINT, PAGE or LINE specified for non-PRINT file.

Informational: The PAGE and LINE options of the PUT statement are valid only for files that are opened with the PRINT attribute.

User Action: Verify that the file is a stream output file and, if so, add PRINT to the file's attribute list and recompile.

NOTREC, Not a RECORD file.

Informational: A record I/O statement (READ, WRITE, DELETE, or REWRITE) was specified for a file that has the STREAM attribute.

User Action: Correct the file's attribute list or use a stream I/O statement to process the file and recompile.

NOTRELSQL, Not a RELATIVE or SEQUENTIAL file

Fatal: The LOCK_NONEXISTENT record option may only be used with relative or sequential files.

User Action: Remove the LOCK_NONEXISTENT option or use a different file organization.

NOTSQL, Not a SEQUENTIAL file.

Informational: A sequential READ or WRITE statement was specified for a file that has the DIRECT attribute. The I/O statement must specify a KEY or KEYFROM option.

User Action: Decide whether the file was to be accessed sequentially or directly, and correct the I/O statement that caused the error.

NOTSTREAM, Stream I/O attempted on RECORD file.

Informational: A GET or PUT statement was used to process a file that has the RECORD attribute.

User Action: Correct the file's attribute list, remembering that certain file description attributes imply the RECORD attribute, and recompile the program.

NOTUPDATE, Attempting to REWRITE or DELETE an INPUT or OUTPUT file.

Informational: The REWRITE and DELETE statements are not valid for files that are opened with either the INPUT or OUTPUT attributes; the file must have the UPDATE attribute.

User Action: Correct the file's attribute list and recompile.

NOVIRMEM, Virtual memory overflow.

Informational: The run-time system attempted to allocate virtual memory for an I/O buffer in the program, but there was insufficient virtual memory available.

User Action: Simplify the source program so that it requires less space.

ONCNVPOS, The erroneous character is at position *entity*.

Informational: This message displays an up arrow below the character in error in the ONSOURCE message.

User Action: Use the information to correct the source field of the conversion.

ONSNOTMOD, ONSOURCE value not modified.

Informational: A normal return occurred from a CONVERSION condition, which would normally result in a retry of the conversion. However, the ONSOURCE value was not modified, so the ERROR condition was raised to prevent an infinite loop.

User Action: Change the ON unit handling the condition to modify the ONSOURCE value using either the ONSOURCE or ONCHAR pseudovariables.

ONSOURCE, The conversion source is *entity*.

Informational: This displays the source string.

User Action: None.

OPEN, Open failure.

Informational: An attempt was made to open a file implicitly by an I/O statement, but the file could not be opened. The UNDEFINEDFILE condition was signaled, but the ON-unit did not successfully open the file.

User Action: Write an ON-unit to handle the UNDEFINEDFILE condition for the given file to ensure that the file will be opened.

PAGESIZE, Invalid PAGESIZE specified.

Informational: The value specified in the PAGESIZE option exceeds the implementation's limit of 32767 or the value is not a positive integer.

User Action: Correct the value specified in the PAGESIZE option.

PAGOVRFLO, Page number overflow.

Informational: The number of pages in a stream file exceeds the implementation's limit of 32767.

User Action: Write an ON-unit using the VAXCONDITION or ANYCONDITION to handle the condition.

PARM, 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.

PROMPTOBIG, PROMPT option too long. Must be less than 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.

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.

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.

RECIDKEY, RECORD_ID_FROM conflicts with KEY 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.

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, or an area is too small to hold the extent of an area being read into it.

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.

RECORDCND, PL/I compiler/run-time error. Please submit an SPR.

Fatal: 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.

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.

RMSF, PL/I internal FAB condition.

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.

RMSR, PL/I internal RAB condition.

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.

SIZE, PL/I compiler/run-time error. Please submit an SPR.

Fatal: 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.

STORAGE, PL/I STORAGE condition.

Fatal: A failure status was returned by LIB$GET_VM for an ALLOCATE statement. The secondary condition value indicates the reason for the failure.

User Action: Correct the situation that caused the condition to be raised.

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. If necessary, use more than one stream I/O statement.

STRRANGE, PL/I STRINGRANGE condition.

Fatal: The third operand in a reference to a SUBSTR built-in function or pseudovariable, or the third parameter in a reference to an INDEX, SEARCH or VERIFY built-in function is beyond the range of the string. This message is only issued for checks that fail in in-line code if the procedure containing this reference was compiled with the /CHECK qualifier. This condition is always checked by PL/I run-time library routines.

User Action: Correct the reference.

STRSIZE, PL/I STRINGSIZE condition.

Fatal: 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.

SUBRANGE, Subscript range check error.

Informational: 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.

SUBRANGE1, Subscript 1 range check error.

Informational: The first subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE2, Subscript 2 range check error.

Informational: The second subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE3, Subscript 3 range check error.

Informational: The third subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE4, Subscript 4 range check error.

Informational: The fourth subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE5, Subscript 5 range check error.

Informational: The fifth subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE6, Subscript 6 range check error.

Informational: The sixth subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE7, Subscript 7 range check error.

Informational: The seventh subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRANGE8, Subscript 8 range check error.

Informational: The eighth subscript in an array reference specifies a value that is beyond the bounds of that dimension. This message is issued only if the procedure containing the reference was compiled with the /CHECK qualifier.

User Action: Correct the reference.

SUBRG, PL/I SUBSCRIPTRANGE condition.

Fatal: 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 2 of a SUBSTR is out of range.

Informational: 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 3 of a SUBSTR is out of range.

Informational: 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.

TRANSMIT, PL/I compiler/run-time error. Please submit an SPR.

Fatal: 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.

UNDFILE, PL/I UNDEFINEDFILE condition on file *entity*.

Fatal: This message is followed by one or more messages that indicate the specific error that occurred during opening of 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, *entity*.

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.

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.

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 UPDATE attribute or with the OUTPUT attribute and ENVIRONMENT(APPEND).

User Action: Correct the source program.

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.

A.3 %DICTIONARY Error Messages

When an error occurs during use of the Common Data Dictionary (CDD), it is generated by one of the following:

- The PL/I compiler, which generates error messages that begin with %PLIG. These messages appear in Section A.1.
- The Common Data Dictionary, which generates error messages that begin with %CDD. These messages appear in the *VAX CDD/Plus Utilities Reference Manual*. CDDL error messages appear in the *VAX CDD/Plus*.
- The CRX, which generates error messages that begin with %CRX. These messages are given in Table A–1.

Informational messages do not inhibit the production of an object file, but may indicate that your results might not be as you had anticipated.

Most error messages indicate that there is an error that cannot be corrected by the user. Therefore, it is requested that you submit an SPR to the CDD or to the product that created the record description.

Table A–1 CRX Error Messages

CRX Error	Message	User Action
%CRX-E-BADBASE	Field description specifies base other than 2 or 10.	Correct the description to be base 2 or 10.
%CRX-E-BADCORLEV	Record description specifies unsupported core level.	Submit SPR to CDD or to the product that created the description.

(continued on next page)

Table A–1 (Cont.) CRX Error Messages

CRX Error	Message	User Action
%CRX-E-BADDIGITS	Field description specifies improper number of digits.	Correct the field description to specify the proper number of digits.
%CRX-E-BADFORMAT	Record description specifies improper record format.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADLENGTH	Field description specifies improper length.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADOCCURS	Dimension description improperly specifies Minimum Occurs.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADOFFSET	Field description specifies improper offset.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADOVERLAY	Field description specifies overlay for nonoverlay field.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADPRTCL	Path name does not designate a node with record protocol.	Correct the path name.
%CRX-E-BADREFER	Field description specifies reference for nonpointer field.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADSCALE	Field description specifies scale greater than precision.	Correct the precision or scale specified in the field description.
%CRX-E-BADSTRIDE	Dimension description specifies improper stride.	Submit SPR to CDD or to the product that created the description.
%CRX-E-BADTAGVAR	Field description specifies tag for nonoverlay field.	Submit SPR to CDD or to the product that created the description.
%CRX-I-INITVAL	Initial value in field description being ignored.	No action.
%CRX-I-LITERALS	Literal definitions in record description being ignored.	No action.
%CRX-E-MEMBADTYP	Field description specifies data type for field with members.	Submit SPR to CDD or to the product that created the description.
%CRX-I-NOCONTIN	Improper continuation after a noncontinuable condition.	Submit a PL/I SPR.
%CRX-E-NOCORATT	Record description does not specify core level.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOFORMAT	Record description does not specify record format.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOLENGTH	Field description does not specify length.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOLOWER	Dimension description does not specify lower bound.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOOFFSET	Field description does not specify offset.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOOVERLAY	Field description does not specify overlay for overlay field.	Submit SPR to CDD or to the product that created the description.

(continued on next page)

Table A–1 (Cont.) CRX Error Messages

CRX Error	Message	User Action
%CRX-E-NOSTRIDE	Dimension description does not specify stride.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOTCOMPUT	Field definition specifies numeric attributes for nonnumeric data.	Submit SPR to CDD or to the product that created the description.
%CRX-E-NOUPPER	Dimension description does not specify upper bound.	Submit SPR to CDD or to the product that created the description.
%CRX-I-REFERENCE	Reference in overlay description being ignored.	No action.
%CRX-I-TAGVALUES	Tag values in overlay description being ignored.	No action.
%CRX-E-UNALIGNED	Field description specifies improper field alignment.	Correct the field description to specify the proper alignment.
%CRX-I-UNKFACIL	Unknown facility specified for record description extraction.	Submit a PL/I SPR.

B

Correspondence of PL/I and RMS

Table B–1 lists the PL/I for OpenVMS VAX and PL/I for OpenVMS AXP ENVIRONMENT options and gives the VAX Record Management Services (RMS) macro, field, or bit setting, as appropriate, that corresponds to each.

For detailed descriptions of the RMS fields, see the *OpenVMS Record Management Services Reference Manual*.

Table B–1 RMS Fields for PL/I ENVIRONMENT Options

Option	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
FILE_ID	n/a
FILE_ID_TO	n/a
FILE_SIZE	$FAB ALQ
FIXED_CONTROL_SIZE	$FAB FSZ
FIXED_CONTROL_SIZE_TO	$FAB RFM=VFC
FIXED_LENGTH_RECORDS	$FAB RFM=FIX
GROUP_PROTECTION	$XABPRO

(continued on next page)

Table B–1 (Cont.) RMS Fields for PL/I ENVIRONMENT Options

Option	RMS Macro Field
IGNORE_LINE_MARKS	n/a
INDEX_NUMBER	$RAB KRF
INDEXED	$FAB ORG=IDX
INITIAL_FILL	$RAB ROP=LOA
MAXIMUM_RECORD_NUMBER	$FAB MRN
MAXIMUM_RECORD_SIZE	$FAB MRS
MULTIBLOCK_COUNT	$RAB MBC
MULTIBUFFER_COUNT	$RAB MBF
NO_SHARE	$FAB SHR=NIL
OWNER_GROUP	$XABPRO UIC
OWNER_MEMBER	$XABPRO UIC
OWNER_PROTECTION	$XABPRO PRO
PRINTER_FORMAT	$FAB RAT=PRN
READ_AHEAD	$RAB ROP=RAH
READ_CHECK	$FAB FOP=RCK
RECORD_ID_ACCESS	$RAB FAC=RFA
RETRIEVAL_POINTERS	$FAB RTV
REWIND_ON_CLOSE	$FAB FOP=RWC
REWIND_ON_OPEN	$FAB FOP=RWO
SCALARVARYING	n/a
SHARED_READ	$FAB SHR=GET
SHARED_WRITE	$FAB SHR=PUT,- GET,UPD,DEL
SPOOL	$FAB FOP=SPL
SUPERSEDE	$FAB FOP=SUP,- NEF,^MXV,^CIF $RAB ROP=^EOF
SYSTEM_PROTECTION	$XABPRO PRO
TEMPORARY	$FAB FOP=TMP
TRUNCATE	$FAB FOP=TEF
WORLD_PROTECTION	$XABPRO PRO
WRITE_BEHIND	$RAB ROP=WBH
WRITE_CHECK	$FAB FOP=WCK

C

Optional Programming Productivity Tools

This appendix provides an overview of optional programming productivity tools. These tools are not included with the PL/I for OpenVMS VAX or PL/I for OpenVMS AXP software; they must be purchased separately. Using these tools can increase your productivity as a PL/I programmer. For information on how to purchase these tools, contact your Digital sales representative.

C.1 Using LSE with PL/I

The Language-Sensitive Editor (LSE) is a powerful and flexible text editor designed specifically for software development. LSE has important features that help you produce syntactically correct code in PL/I for OpenVMS VAX and PL/I for OpenVMS AXP.

To invoke LSE, specify the LSEDIT command followed by a file name with a PLI file type at the DCL prompt. For example:

```
$ LSEDIT USER.PLI
```

The following sections describe some of the key features of LSE. Section C.1.1 discusses how to enter source code using LSE, and Section C.1.2 describes LSE's compiler interface features. Section C.1.3 gives examples of how to generate PL/I source code with LSE.

For more details on advanced features of LSE and SCA, see the *Guide to Language-Sensitive Editor for VMS Systems* and the *Guide to Source Code Analyzer for VMS Systems*.

C.1.1 Entering Source Code Using Tokens and Placeholders

LSE's language-sensitive features simplify the tasks of developing and maintaining software systems. These features include language-specific placeholders and tokens, aliases, comment and indentation control, and templates for subroutine libraries. The following sections describe these features in detail.

LSE can be used as a traditional text editor. In addition, you can have the power of using LSE's tokens and placeholders to step through each program construct and supply text for those constructs needing it.

Placeholders are markers in the source code that indicate locations where you can provide program text. These placeholders help you to supply the appropriate syntax in a given context. Generally, you do not need to enter placeholders; rather, they are inserted for you by LSE. Placeholders are surrounded by brackets or braces.

The types of LSE placeholders are as follows:

Type	Description
Terminal placeholders	Provide text strings that describe valid replacements for the placeholder
Nonterminal placeholders	Expand into additional language constructs
Menu placeholders	Provide a list of options corresponding to the placeholder

Placeholders are either optional or required. Required placeholders, indicated by braces, represent places in the source code where you must provide program text. Optional placeholders, indicated by brackets, represent places in the source code where you can either provide additional constructs or erase the placeholder.

You can move forward or backward from placeholder to placeholder. In addition, you can delete or expand placeholders as needed.

Tokens typically represent keywords in PL/I. Tokens are provided for all PL/I statements, built-in functions, and built-in subroutines. When expanded, tokens provide additional language constructs. You can enter tokens directly into the buffer.

Generally, you use tokens in situations, such as modifying an existing program, where you want to add additional language constructs and there are no placeholders. For example, typing IF and issuing the EXPAND command causes a template for an IF construct to appear on your screen.

You can use tokens to insert text when editing an existing file by typing the name for a function or keyword and issuing the EXPAND command. You can also use tokens to bypass long menus in situations where expanding a placeholder, such as [statement], would result in a lengthy menu.

LSE provides commands that allow you to manipulate tokens and placeholders. These commands and their default key bindings are as follows:

Command	Key Binding	Function
EXPAND	Ctrl/e	Expands a placeholder
UNEXPAND	PF1-Ctrl/e	Reverses the effect of the most recent placeholder expansion
GOTO PLACEHOLDER/FORWARD	Ctrl/n	Moves the cursor forward to the next placeholder
GOTO PLACEHOLDER/REVERSE	Ctrl/p	Moves the cursor backward to the previous placeholder
ERASE PLACEHOLDER/FORWARD	Ctrl/k	Erases a placeholder
UNERASE PLACEHOLDER	PF1-Ctrl/k	Restores the most recently erased placeholder
None	Down arrow	Moves the indicator through a screen menu toward the bottom
None	Up arrow	Moves the indicator through a screen menu toward the top
None	{ ENTER RETURN }	Selects a menu option

To display a list of all the defined tokens provided by PL/I, enter the SHOW TOKEN command as follows:

```
LSE> SHOW TOKEN
```

To display a list of all the defined placeholders provided by PL/I, enter the SHOW PLACEHOLDER command as follows:

```
LSE> SHOW PLACEHOLDER
```

You must have a PLI file in your current buffer in order to use the SHOW TOKEN or SHOW PLACEHOLDER command. To put a copy of either list into a separate file, first enter the appropriate SHOW command to put the list into the $SHOW buffer. Then enter the following commands:

```
LSE> GOTO BUFFER $SHOW
LSE> WRITE filename
```

To obtain a hard copy of the list, use the PRINT command at the DCL level to print the file you created.

To obtain information about a particular token or placeholder, you can also specify a token name or placeholder name after the SHOW TOKEN or SHOW PLACEHOLDER command.

C.1.2 Compiling Source Code

To compile your code and to review compilation errors without leaving the editing session, you can use the LSE commands COMPILE and REVIEW. The COMPILE command issues a DCL command in a subprocess to invoke the PL/I compiler. The compiler then generates a file of compile-time diagnostic information that LSE can use to review compilation errors. The diagnostic information is generated with the /DIAGNOSTICS qualifier that LSE appends onto the compilation command.

For example, if you issue the COMPILE command while in the buffer USER.PLI, the resulting DCL command is as follows:

```
$ PLI USER.PLI/DIAGNOSTICS=USER.DIA
```

LSE supports all of the PL/I compiler's command qualifiers as well as user-supplied command procedures. You can specify DCL qualifiers, such as the /LIBRARY qualifier, when invoking the compiler from LSE. For example, to generate Source Code Analyzer (SCA) data, you can use the following command:

```
LSE> COMPILE $/ANALYSIS_DATA
```

The REVIEW command displays any diagnostic messages that result from a compilation. LSE displays the compilation errors in one window and the corresponding source code in a second window. This multiwindow capability allows you to review your errors while examining the associated source code. This capability eliminates tedious steps in the error correction process, and helps ensure that all the errors are fixed before you compile your program again.

LSE provides several commands to help you review errors and examine your source code. The following table lists these commands and their default key bindings where applicable.

Command	Key Binding	Function
COMPILE	None	Compiles the contents of the source buffer

Command	Key Binding	Function
COMPILE /REVIEW	None	Compiles the contents of the source buffer, puts LSE into REVIEW mode, and displays any errors resulting from the compilation
REVIEW	None	Performs the same function as the /REVIEW qualifier on the COMPILE command: puts LSE into REVIEW mode, and displays any errors resulting from the last compilation
END REVIEW	None	Removes the buffer $REVIEW from the screen; returns the cursor to a single window containing the source buffer
GOTO SOURCE	Ctrl/g	Moves the cursor to the source buffer that contains the error
NEXT STEP	Ctrl/f	Moves the cursor to the next error in the buffer $REVIEW
PREVIOUS STEP	Ctrl/b	Moves the cursor to the previous error in the buffer$REVIEW
None	{ Down arrow / Up arrow }	Moves the cursor within a buffer

C.1.3 Examples

This section describes the special features of PL/I available through LSE and provides examples of PL/I code written with LSE.

The following examples show expansions of the more frequently used PL/I for OpenVMS VAX and PL/I for OpenVMS AXP tokens and placeholders. The examples are expanded to show the formats and guidelines LSE provides; however, not all of the examples are fully expanded.

The examples show expansions of the following PL/I for OpenVMS VAX and PL/I for OpenVMS AXP features:

- DO Statement
- IF Statement
- Assignment Statement
- DECLARE Statement
- SUBSTR Expression
- %PROCEDURE Statement

Instructions and explanations precede each example, and an arrow (→) indicates the line in the code where an action has occurred.

To reproduce the examples, invoke LSE and the PL/I language by using the following syntax:

```
LSEDIT [/qualifier . . . ] filename.PLI
```

See Section C.1.1 for the commands that manipulate tokens and placeholders.

When you use LSE to create a new PL/I program, the initial string appears at the top of the screen as follows:

```
[program]
```

Expand the placeholder [program] to produce the following:

```
        /*
        [module_header_comments]
        **/
        [preprocessor_statement] . . .
        [declare_statement] . . .
        {procedure} . . .
```

Erase the first five lines of this expansion and expand the placeholder {procedure}.

```
        /*
        [procedure_header_comments]
        **/
-->     {entry_name}: procedure [parameters] [options] [returns] [recursive];
        [declare_statement] . . .
        [statement] . . .
        end {entry_name};
        [procedure] . . .
```

Erase the first three lines. Enter `test` over the placeholder {entry_name}. (Once the cursor is moved from that text, LSE automatically fills in the next occurrence of {entry_name}.) Erase the placeholder [parameters].

```
-->     test: procedure [options] [returns] [recursive];
        [declare_statement] . . .
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [options].

```
-->     test: procedure options({option} . . .  ) [returns] [recursive];
        [declare_statement] . . .
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the list placeholder {option} to produce a menu and select the option MAIN.

```
-->     test: procedure options(main, [option] . . .  ) [returns] [recursive];
        [declare_statement] . . .
        [statement] . . .
        end test;
        [procedure] . . .
```

Erase the duplicated placeholder [option], and the placeholders [returns], [recursive], and [declare_statement].

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Each of the following examples starts from this expansion.

C.1.4 DO Statement

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the list placeholder [statement] to display a menu and select the option DO.

```
        test: procedure options(main);
-->     do [do_specification];
            [statement] . . .
            end;
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [do_specification] to display a menu and select the option `while ({boolean_expression})`.

```
        test: procedure options(main);
-->     do while({boolean_expression});
            [statement] . . .
            end;
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter `'1'b` over the placeholder {boolean_expression}.

```
        test: procedure options(main);
-->     do while('1'b);
            [statement] . . .
            end;
        [statement] . . .
        end test;
        [procedure] . . .
```

C.1.5 IF Statement

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the list placeholder [statement] to display a menu and select the option IF.

```
        test: procedure options(main);
-->     if {boolean_expression}
        then
            [if_action]
        [else_clause]
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter `not_control` over the placeholder {boolean_expression}.

```
        test: procedure options(main);
-->     if not_control
        then
            [if_action]
        [else_clause]
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter `string='missile';` over the placeholder [if_action].

```
        test: procedure options(main);
        if not_control
        then
-->         string='missile';
        [else_clause]
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [else_clause].

```
        test: procedure options(main);
        if not_control
        then
            string='missile';
-->     else
            {if_action}
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter `string='control';` over the placeholder {if_action}.

```
        test: procedure options(main);
        if not_control
        then
            string='missile';
        else
-->         string='control';
        [statement] . . .
        end test;
        [procedure] . . .
```

C.1.6 Assignment Statement

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the list placeholder [statement] to display a menu and select the option ASSIGNMENT.

```
        test: procedure options(main);
-->     {target_variable} . . .   ={expression};
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder {target_variable} to display a menu and select the option {SCALAR_VARIABLE}.

```
        test: procedure options(main);
-->     {SCALAR_VARIABLE},[target_variable] . . .   ={expression};
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter general over the placeholder {SCALAR_VARIABLE} and erase the placeholder [target_variable]. Enter 'Lee' over the placeholder {expression}.

```
        test: procedure options(main);
-->     general='Lee';
        [statement] . . .
        end test;
        [procedure] . . .
```

C.1.7 DECLARE Statement

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [statement] to display a menu and select the option DECLARE.

```
        test: procedure options(main);
-->     declare
            {declaration} . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the list placeholder {declaration} to display a menu and select the option {NON_STRUCTURE_DECLARATION}.

```
        test: procedure options(main);
        declare
-->         {NON_STRUCTURE_DECLARATION} . . .  ,
            [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder {NON_STRUCTURE_DECLARATION} to display a menu and select the option {SIMPLE_DECLARATION}.

```
        test: procedure options(main);
        declare
-->         {identifier}[array_bounds] [datatype] [storage_class],
            [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter what_if over the placeholder {identifier} and erase the placeholder [array_bounds].

```
        test: procedure options(main);
        declare
-->         what_if [datatype] [storage_class],
            [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [datatype] to display a menu and select the option [BIT_DATATYPE].

```
        test: procedure options(main);
        declare
-->        what_if bit[string_length_opt] [aligned] [storage_class],
           [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Erase the placeholder [string_length_opt] and expand the placeholder [aligned].

```
        test: procedure options(main);
        declare
-->        what_if bit aligned [storage_class],
           [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [storage_class] to display a menu and select the option [CONTROLLED].

```
        test: procedure options(main);
        declare
-->        what_if bit aligned controlled [external],
           [declaration] . . .  ;
        [statement] . . .
        end test;
        [procedure] . . .
```

Erase the placeholders [external] and [declaration].

```
        test: procedure options(main);
        declare
-->         what_if bit aligned controlled;
        [statement] . . .
        end test;
        [procedure] . . .
```

C.1.8 SUBSTR Expression

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder [statement] to produce a menu and select the option ASSIGNMENT.

```
        test: procedure options(main);
-->     {target_variable} . . .   ={expression};
        [statement] . . .
        end test;
        [procedure] . . .
```

Expand the placeholder {target_variable} to display a menu and select the option {SCALAR_VARIABLE}.

```
        test: procedure options(main);
-->     {SCALAR_VARIABLE},[target_variable] . . .   ={expression};
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter vhf over the placeholder {SCALAR_VARIABLE}. Erase the placeholder [target_variable]. Enter substr over the placeholder {expression} and expand substr.

```
        test: procedure options(main);
-->     vhf=substr({string_expression},{position}[length_option]);
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter 'where''s dixie' over the placeholder {string_expression}.

```
        test: procedure options(main);
-->     vhf=substr('where''s dixie',{position}[length_option]);
        [statement] . . .
        end test;
        [procedure] . . .
```

Enter the value 9 over the placeholder {position}. Expand the placeholder [length_option], and enter the value 5 over the placeholder {integer_expression}.

```
        test: procedure options(main);
-->     vhf=substr('where''s dixie',9,5);
        [statement] . . .
        end test;
        [procedure] . . .
```

C.1.9 %PROCEDURE Statement

Begin at the following expansion developed in Section C.1.3.

```
        test: procedure options(main);
        [statement] . . .
        end test;
        [procedure] . . .
```

Erase the list placeholder [statement]. Enter %proc and expand it.

```
        test: procedure options(main);
-->     %{identifier}: procedure [parameters] [statement_option]
                returns({prep_attribute});
            [prep_proc_statements] . . .
            return({prep_expression});
            %end;
        end test;
        [procedure] . . .
```

Enter f over the placeholder {identifier}.

```
        test: procedure options(main);
-->     %f: procedure [parameters] [statement_option]
                returns({prep_attribute});
            [prep_proc_statements] . . .
            return({prep_expression});
            %end;
        end test;
        [procedure] . . .
```

Erase the placeholders [parameters] and [statement_option].

```
        test: procedure options(main);
-->     %f: procedure returns({prep_attribute});
            [prep_proc_statements] . . .
            return({prep_expression});
            %end;
        end test;
        [procedure] . . .
```

Enter char over the placeholder {prep_attribute} and erase the placeholder [prep_proc_statements].

```
        test: procedure options(main);
-->     %f: procedure returns(char);
            return({prep_expression});
            %end;
        end test;
        [procedure] . . .
```

Enter time( ) over the placeholder {prep_expression}.

```
        test: procedure options(main);
        %f: procedure returns(char);
-->         return(time());
            %end;
        end test;
        [procedure] . . .
```

C.2 Using the Source Code Analyzer

The Source Code Analyzer (SCA) is an interactive source code cross-reference and static analysis tool that works with most OpenVMS VAX and OpenVMS AXP programming languages. SCA helps developers keep track of the details of complex, large-scale software systems by displaying source information in response to user queries. SCA uses data generated by the PL/I compiler to supply the requested source information. That information is stored in a unique location, the SCA library. The data in an SCA library consists of the names of, and information about, all the symbols, modules, and files encountered during a specific compilation of the source.

SCA has both **cross-reference** and **static analysis** query features. Cross-referencing supplies information about program symbols and source files. Cross-referencing features include the following:

- Locating names, and occurrences (uses) of these names
- Querying a specified set of names or partial names (with wildcards allowed)
- Limiting a query to specific characteristics (such as routine names, variable names, or source files)
- Limiting a query to specific occurrences (such as the primary declaration of a symbol, read or write occurrences of a symbol, or occurrences of a file)

The static analysis query features of SCA provide structural information on the interrelation of routines, symbols and files. Static analysis features include the following:

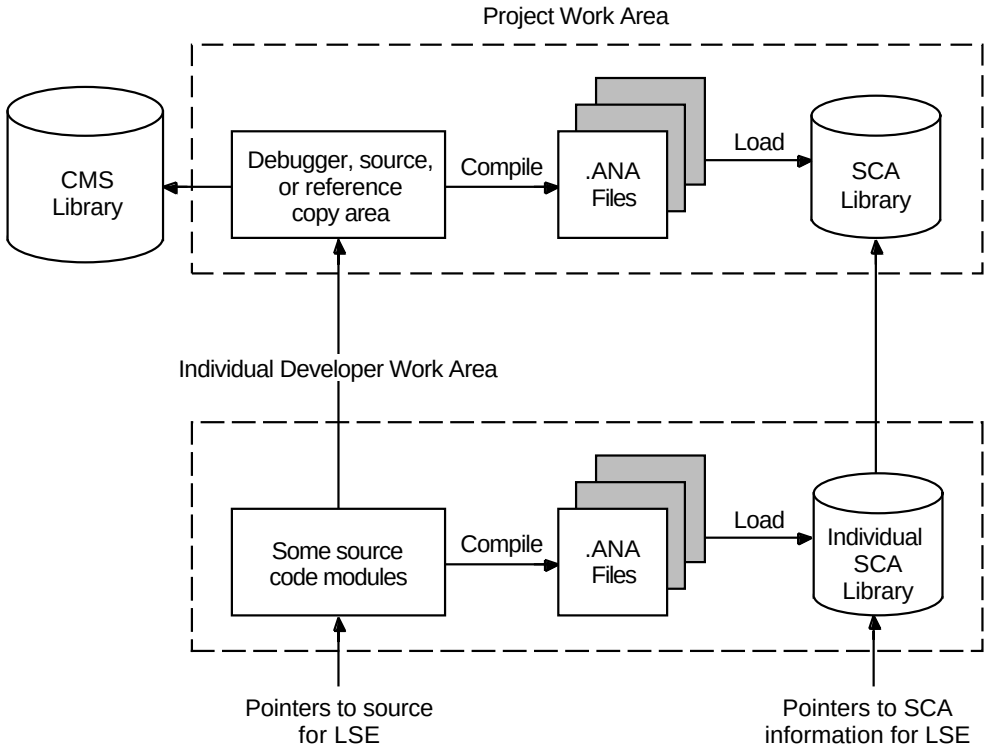
- Displaying routine calls to and from a specified routine
- Analyzing routine calls for consistency as to the numbers and data types of arguments passed, and the types of values returned

SCA is fully integrated with LSE to provide extended features. By using SCA with LSE, you can view any portion of an entire system and edit related source files.

C.2.1 Multimodular Development

The cross-referencing and static analysis features of SCA can become useful during the implementation and maintenance phases of a project that involves many programming modules. For example, Figure C–1 shows a project team work area that contains a set of source modules. To keep track of these modules in their various development stages, the team can use a code management tool, such as Code Management System (CMS), which is represented in the figure by the CMS Library.

Figure C–1 Use of SCA for Multimodular Development



When the team compiles the source code, a /ANALYSIS_DATA qualifier to the COMPILE command instructs the PL/I compiler to generate SCA-required source information (.ANA data files) from the sources. The team then instructs SCA to load the .ANA files into a previously established SCA Library.

When a team member wants to do additional development work on specific modules, that member sets up an individual work area. Such individual work areas might consist of the following:

- Copies of source and object modules from the project libraries.

- Local SCA libraries that contain copies of the module information required to complete assigned tasks.

To make available the module-viewing capabilities of SCA/LSE integration, the project team member must inform LSE of the locations of latest sources, and the related source information. The team member provides pointers to these locations by supplying a search list for LSE. The search list first points to source modules in individual team members' default directories, and then points to the remaining modules in the project source directory. With such an arrangement, each member can effectively see through the local work area to the project-wide area. If an individual work area contains only new modules, and all of the work can be done with local resources, the team member need not specify the pointers to the project-wide area.

The following sections provide a general overview of SCA and discuss some of the commands that are available to you while using SCA within LSE. For detailed information on SCA and its use with various programming languages, refer to the manual *Using VAXset*.

C.2.2 Setting Up an SCA Environment

To set up an SCA environment, you must take the following steps:

1. Create an SCA library in a subdirectory.
2. Use the PL/I compiler to generate the data analysis (.ANA) files for each source module in your system.
3. Load these data analysis files into your local SCA library.

You are then ready to use SCA to conduct source information queries.

C.2.2.1 Creating an SCA Library

To use SCA, you must have an SCA library to store the detailed source analysis data that the PL/I compiler collects. Source analysis data is information about all of the symbols, files and modules contained in the source.

To create an SCA library you first create a subdirectory at the DCL level. For example:

```
$ CREATE/DIRECTORY PROJ:[USER.LIB1]
```

This command creates a subdirectory LIB1 for a local SCA library.

To initialize a new SCA library you specify the CREATE LIBRARY command. This command has the following form:

```
CREATE LIBRARY [/qualifier . . . ] directory-spec[, . . . ]
```

For example:

```
$ SCA CREATE LIBRARY [.LIB1]
```

This command initializes and activates library LIB1.

C.2.2.2 Generating the Data Analysis Files

SCA uses detailed source data that is generated by the PL/I compiler. When you specify the /ANALYSIS_DATA qualifier on the PLI command, the generated data is output to a file with the default type .ANA. For example:

```
$ PLI/LIST/DIAGNOSTICS/ANALYSIS_DATA PG1,PG2,PG3
```

This command line compiles the input files PG1.PLI, PG2.PLI, and PG3.PLI, and generates four corresponding output files for each input file with the file types OBJ, LIS, DIA, and ANA. SCA puts these files in your current default directory unless you specify otherwise.

C.2.2.3 Loading Data Analysis Files into a Local Library

Before you can examine the information in the compiler-generated source analysis (.ANA) files, you must load the files into an SCA library using the LOAD command. The LOAD command has the following form:

```
LOAD [/qualifier . . . ] file-spec[, . . . ]
```

For example:

```
LSE> LOAD PG1,PG2,PG3
```

This command loads your library with the modules contained in the data analysis files PG1, PG2, and PG3.

C.2.2.4 Selecting an SCA Library

To select an existing SCA library to use with your current SCA session, use the SET LIBRARY command. The command has the following form:

```
SET LIBRARY [/qualifier . . . ] directory-spec[, . . . ]
```

A message appears in the message buffer at the bottom of your screen, indicating whether your SCA library selection succeeded.

C.2.3 Using SCA for Cross-Referencing

Once you have set up your SCA environment, you can ask for symbol or file information by using the SCA command FIND. The FIND command has the following form:

```
FIND [/qualifier . . . ] [name-expression[, . . . ]]
```

name-expression
The name-expression can be explicit (for example, ABC) or can contain wildcards (for example, ABC* or AB%).

For example:

```
LSE> FIND ABC,XY%
```

You can query an SCA library for the following:

Name	A series of characters that uniquely identifies a symbol or a file.
Item	An appearance of a symbol (such as a variable, constant, label, or procedure) or a file.
Occurrence	The use of a symbol or a file.

To limit the information resulting from a query, you can use qualifiers on the FIND command, such as the /DECLARATIONS and /REFERENCE qualifiers. For example:

```
LSE> FIND/REFERENCES=CALL BUILD_TABLE
```

This command causes SCA to report only references in the source code where the routine BUILD_TABLE is called.

When you first issue a FIND command within LSE, you initiate a **query session**. Within this context, the integration of LSE and SCA provides the following commands that can be used only within LSE:

{ NEXT } { PREVIOUS }	Closely associated commands that let you step through one or more query buffer displays within LSE.
{ NAME ITEM OCCURRENCE QUERY STEP }	
GOTO SOURCE	Displays the source corresponding to the current query item.
GOTO DECLARATION	Positions the cursor on a symbol declaration in one window, and displays the source code that contains the symbol declaration in another window.

D

Rules for Conversion of Data

This appendix provides details of the data type conversions that PL/I performs when assigning values to variables. The rules for conversions apply to the following:

- Assignment statements
- Arguments passed to a procedure
- Values specified in a RETURN statement
- Arguments converted by the built-in functions FIXED, FLOAT, BINARY, DECIMAL, BIT, or CHARACTER
- Character-string arguments to the PUT and GET statements

D.1 Assignments to Arithmetic Variables

You can assign expressions of any computational type to arithmetic variables. Note, however, that the compiler may issue a warning message unless an explicit conversion function is used. The conversion rules are described in the following sections for each source type.

D.1.1 Arithmetic to Arithmetic Conversions

You can assign a source expression of any arithmetic type to a target variable of any arithmetic type. Note the following qualifications:

- If the target is a variable of type FIXED BINARY or FIXED DECIMAL, then the FIXEDOVERFLOW condition is signaled when the source value has a larger number of integral digits than are specified in the precision of the target. If the target is a fixed-point binary variable, FIXEDOVERFLOW is signaled if the source value exceeds the storage allocated for the target.
- If the target is a variable of type FIXED-POINT(p,q) and the source value has more than q fractional digits, then the excess fractional digits of the source are truncated, and no condition is signaled. If the source has fewer than q fractional digits, the source value is padded on the right with zeros.
- If the target value is floating point and the absolute source value is too large to be represented by an OpenVMS VAX or OpenVMS AXP floating-point type (see Chapter 10), then the OVERFLOW condition is signaled, and the value of the target is undefined. If the absolute source value is too small to be represented, the value zero is assigned to the target if the UNDERFLOW option is not enabled. If the UNDERFLOW option is enabled, the UNDERFLOW condition is signaled, and the value of the target is undefined.

D.1.2 Pictured to Arithmetic Conversions

In PL/I for OpenVMS VAX and PL/I for OpenVMS AXP, all pictured values have the associated attributes FIXED DECIMAL(p,q), where p is the total number of characters in the picture specification that specify decimal digits, and q is the total number of these digits that occur to the right of the V character. If the picture specification does not include a V character, then q is zero. This associated fixed-point decimal value is assigned to the target, following the rules for arithmetic to arithmetic conversion described in Section D.1.1.

D.1.3 Bit-String to Arithmetic Conversions

When a bit-string value is assigned to an arithmetic variable, PL/I treats the bit string as a nonnegative fixed-point binary value. If the converted value is greater than or equal to 2^{31} for OpenVMS VAX or 2^{63} for OpenVMS AXP, then FIXEDOVERFLOW is signaled. The leftmost bit in the bit string (as output by PUT LIST) is the most significant bit in the fixed-point binary value, not its sign. If the bit string is null, the fixed-point binary value is zero. The intermediate fixed-point binary value is then converted to the target arithmetic type.

Note that a bit string interpreted as a fixed-point binary value changes its value when assigned to a bit-string variable of a different length. See the *PL/I for OpenVMS Systems Reference Manual* for further details.

D.1.4 Character String to Arithmetic Conversions

When a character string is assigned to an arithmetic value, PL/I interprets the string as an arithmetic constant and creates an intermediate numeric value based on the characters in the string. The string can contain any series of characters that describes a valid arithmetic constant. If it contains any invalid characters, the ERROR condition is signaled.

PL/I then converts the intermediate value to the data type of the target, following the rules for arithmetic to arithmetic conversions. In conversions to fixed point, FIXEDOVERFLOW is signaled if the character string specifies too many integral digits. Excess fractional digits are truncated without signaling a condition.

If the source character string is null or contains all spaces, the resulting arithmetic value is zero.

D.2 Assignments to Bit-String Variables

In the conversion of any data type to a bit string, PL/I first converts the source data item to an intermediate bit-string value. Then, based on the length of the target string, it performs one of the following:

- If the length of the target bit-string value is greater than the length of the intermediate string, the target bit string (as represented by PUT LIST) is padded with zeros on the right.
- If the length of the target bit-string value is less than the length of the intermediate string, the intermediate bit string (as represented by PUT LIST) is truncated on the right.

The following sections describe how PL/I arrives at the intermediate bit-string value for each data type.

D.2.1 Arithmetic and Pictured to Bit-String Conversions

In converting an arithmetic value to a bit-string value, PL/I first computes the absolute value of the arithmetic value, and then converts it to an integer of type FIXED BINARY with a maximum precision of 31 for OpenVMS VAX or 63 for OpenVMS AXP. If this conversion results in an integer larger than the data type can accommodate, the FIXEDOVERFLOW condition is signaled; otherwise, each of the bits of the intermediate bit string represents a binary digit of n.

During the conversion, the sign of the arithmetic value and any fractional digits are lost. As a result, a value that contains only fractional digits (such as 0.2312) is converted to an all-zero bit string.

If an arithmetic value is assigned to a bit-string variable, and that variable is assigned to a second variable of different length, the effect is to multiply or divide the arithmetic value as a result of padding or truncating the bit string. See the *PL/I for OpenVMS Systems Reference Manual* for further details.

D.2.2 Character-String to Bit-String Conversions

PL/I can convert a character string of 0s and 1s to a bit string. Any character in the character string other than 0 or 1, including spaces, will signal the ERROR condition. If the source is a null character string, the intermediate string is a null bit string.

D.3 Assignments to Character-String Variables

In the conversion of any data type to a character string, PL/I first converts the source value to an intermediate character-string value. Then it performs one of the following:

- If the length of the intermediate string is zero, a null string is assigned to the target.
- If the target is a returns descriptor with an asterisk extent (as in RETURNS CHAR(*)), the intermediate string is assigned to the target.
- If the intermediate string is shorter than the maximum length of the target, and the target has the VARYING attribute, it is assigned the value of the intermediate string without trailing spaces. If the target does not have the VARYING attribute, the string is padded with trailing spaces.
- If the maximum length of the target character string is less than the length of the intermediate string, the intermediate string is truncated.

The following sections describe how PL/I arrives at the intermediate string for conversion of each data type. Examples show the intermediate and resulting values.

D.3.1 Arithmetic to Character-String Conversions

The manner in which PL/I converts the arithmetic item depends on the data type of the source, as described in the following subsections.

D.3.1.1 Conversion from Fixed-Point Binary or Decimal

If the source value is of type FIXED BINARY, PL/I first converts it to type FIXED DECIMAL. PL/I converts a value with attributes FIXED DECIMAL to an intermediate string with the numeric value right justified in the string. Following is a description of the format of the intermediate string:

- If there are no fractional digits, the first two characters of the string are spaces. The last characters in the string are the digit characters representing all the digits in the integer; leading zeros are replaced by spaces except in the last position. If the integer is negative, a minus sign immediately precedes the first digit; if not, this position contains a space. At least one digit always appears in the last position in the string.
- If there are no integral digits, the first three characters are (in order) an optional minus sign if the fraction is negative, the digit 0, and a decimal point. If the number is not negative, the first character is a space. The last characters in the string are all the fractional digits of the number.
- If there are both integral and fractional digits, the first character is always a space. The last characters are all the fractional digits of the number and are preceded by a decimal point; the decimal point is always preceded by at least one digit, which may be 0; all integral digits appear before the decimal point, and leading zeros are replaced by spaces. A minus sign precedes the first integral digit if the number is negative; if not, then the minus sign is replaced by a space.

These rules may cause confusion if you do not take into account the leading spaces. In the following examples, the letter b represents a space:

```
DECLARE STRING1 CHARACTER (8),
        STRING2 CHARACTER (4);

STRING1 = 283472.;
        /* intermediate string = 'bbb283472',
        STRING1 = 'bbb28347' */

STRING2 = 283472.;
        /* intermediate string = 'bbb283472',
        STRING2 = 'bbb2' */
STRING2 = -283472.;
        /* intermediate string = 'bb-283472',
        STRING2 = 'bb-2' */
STRING2 = -.003344;
        /* intermediate string = '-0.003344',
        STRING2 = '-0.0' */

STRING2 = -283.472;
        /* intermediate string = 'b-283.472',
        STRING2 = 'b-28' */

STRING2 = 283.472;
        /* intermediate string = 'bb283.472',
        STRING2 = 'bb28' */
```

D.3.1.2 Conversion from Floating-Point Binary or Decimal

If the source value is of type FLOAT BINARY, it is converted to FLOAT DECIMAL. For a value of type FLOAT DECIMAL(p), where p is less than or equal to 34, the intermediate string is of length p+6; this allows extra characters for the sign of the number, the decimal point, the letter E, the sign of the exponent, and the 2-digit exponent.

Note

For PL/I for OpenVMS VAX or PL/I for OpenVMS AXP, if the value is a floating-point number of the type G-float, three characters are allocated to the exponent, and the length of the string is p+7. For PL/I for OpenVMS VAX only, if the value is of type H-float, four characters are allocated to the exponent, and the length of the string is p+8.

If the number is negative, the first character is a minus sign; otherwise, the first character is a space. The subsequent characters are a single digit (which may be 0), a decimal point, p–1 fractional digits, the letter E, the sign of the exponent (always + or –), and the exponent digits. The exponent field is of fixed length, and the zero exponent is shown as all zeros in the exponent field.

For example:

```
CONCH: PROCEDURE OPTIONS(MAIN);

DECLARE OUT PRINT FILE;

OPEN FILE(OUT) TITLE('CONCH.OUT');

PUT SKIP FILE(OUT) EDIT('''',25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',1.233325E-5,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-1.233325E-5,'''') (A);

END CONCH;
```

The program CONCH produces the following output:

```
' 2.5E+26'
'-2.5E+26'
' 1.233325E-05'
'-1.233325E-05'
```

The PUT statement converts its output sources to character strings, following the rules described in this section. Note that the output strings have been surrounded with apostrophes to make the spaces distinguishable. Also note that, in each case, the width of the quoted output field (that is, the length of the converted character string) is the precision of the floating-point constant plus 6.

D.3.2 Pictured to Character-String Conversions

If the source value is pictured, its internal, character-string representation is used without conversion as the intermediate character string.

D.3.3 Bit-String to Character-String Conversions

When PL/I converts a bit string to a character string, it converts each bit (as represented by PUT LIST) to a 0 or 1 character in the corresponding position of the intermediate character string.

If the bit string is a null string, the intermediate character string is also null.

D.4 Assignments to Pictured Variables

A source expression of any computational type can be assigned to a pictured variable. The target pictured variable has a precision (p), which is defined as the number of characters in its picture specification that specify decimal digits. It also has a scale factor (q), which is defined as the number of picture characters that specify digits and occur to the right of the V character in the picture specification. If there is no V character, or if all digit-specification characters are to the left of V, then q is zero.

The source expression is converted to a fixed-point decimal value v of precision (p,q), following the rules given in Section D.1 for conversion from the source data type to fixed decimal. This value is then edited to a character string s, as specified by the picture specification, and the value s is assigned to the pictured target.

When the value v is being edited to the string s, the ERROR condition is signaled if the value of v is less than zero and the picture specification does not contain one of the characters S, +, –, T, I, R, CR, or DB. The value of s is then undefined. FIXEDOVERFLOW is also signaled if the value v has more integral digits than are specified by the picture specification of the target.

D.5 Conversions Between Offsets and Pointers

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. Pointer variables are given values by assignment from existing pointer values or from conversion of offset values.

The OFFSET built-in function converts a pointer value to an offset value. The POINTER built-in function converts an offset value to a pointer. These functions are described in the *PL/I for OpenVMS Systems Reference Manual*.

PL/I also automatically converts a pointer value to an offset value, and an offset value to a pointer value, in an assignment statement. The following assignments are valid:

- pointer-variable = pointer-value;
- offset-variable = offset-value;
- pointer-variable = offset-variable;
- offset-variable = pointer-value;

In the latter two cases, the offset variable must have been declared with an area reference.

E

The VAX Common Data Dictionary

This appendix describes the VAX Common Data Dictionary (CDD).

The CDD is a set of shareable data definitions (language-independent structure declarations) that are defined by a system manager or data administrator. The CDD provides a central repository that can be shared and that is protected from unauthorized access. The definitions stored in the CDD help the system manager coordinate an effective data management system.

Using the CDD has two advantages:

- Record declarations are language independent.
- A single declaration helps guarantee the accuracy and reliability of data.

The CDD is one of the many layered products available from Digital, and not all systems that use PL/I for OpenVMS VAX use the CDD. Therefore, PL/I CDD support is only meaningful if the CDD is on your system. If you are not certain, see your system manager.

CDD data definitions are organized hierarchically in much the same way that files are organized in directories and subdirectories. For example, a dictionary for defining personnel data might have separate directories for each employee type. A directory for salespeople might have subdirectories that would include data definitions for records such as salary and commission history or personnel history.

CDD entries are stored as an internal form; descriptions of data definitions are entered into the dictionary in a unique, general-purpose language called Common Data Dictionary Language (CDDL), and the CDDL compiler converts the data descriptions to an internal form, making them independent of any higher-level language. When a program is compiled, CDD data definitions are drawn into higher-level language programs (provided the data attributes are consistent). Program listings include CDD data definitions in the same language as the application program.

The following examples illustrate how data definitions are written for the CDD. The first example is a structure declaration written in CDDL. The second example shows the same structure as it would appear in a PL/I listing.

Example 1

```
PAYROLL_RECORD STRUCTURE.
  SALESPERSON STRUCTURE.
    NAME                  DATATYPE IS TEXT 30.
    ADDRESS               DATATYPE IS TEXT 40.
    SALESPERSON_ID           DATATYPE IS UNSIGNED NUMERIC 5.
  END SALESPERSON STRUCTURE.
```

Example 2

```
DECLARE 1 PAYROLL_RECORD,
          2 SALESPERSON,
            3 NAME CHARACTER(30),
            3 ADDRESS CHARACTER(40),
            3 SALESPERSON_ID PIC '(5)9';
```

The CDD provides two utilities for creating and maintaining a dictionary:

- The Dictionary Management Utility (DMU), for creating and maintaining the CDD's directory hierarchy, history lists, and access control lists
- The Dictionary Verify/Fix Utility (CDDV), for repairing damaged dictionary files

DMU commands create directories and define record paths. Once these paths are established, records can be included and used in PL/I for OpenVMS VAX programs with the %DICTIONARY statement. For a detailed description of the %DICTIONARY statement, see Section 10.2.6 of the *PL/I for OpenVMS Systems Reference Manual*.

At compile time, the CDD record and its attributes are extracted from the designated CDD record node; then the record's corresponding PL/I declaration is entered into the object module.

E.1 PL/I and CDDL Data Types

The CDD supports some data types that are not native to PL/I. If a data definition contains an unsupported data type, PL/I makes the unsupported data type accessible by declaring it as data type BIT_FIELD or BYTE_FIELD. PL/I does not attempt to approximate a data type that is not supported by PL/I. For example, an F_FLOATING_COMPLEX number is declared BYTE_FIELD(8), not (2)FLOAT(24).

However, the use of the BIT_FIELD and BYTE_FIELD types is limited. Data declared with these data types can be manipulated only with the PL/I built-in functions ADDR, INT, POSINT, SIZE, and UNSPEC. Variables declared with BIT_FIELD or BYTE_FIELD can also be passed as parameters provided the parameter is declared as ANY. This limits references to data declared with BIT_FIELD or BYTE_FIELD to contexts in which the interpretation of a data type is not applied to the reference.

For example:

```
/* Declaration supplied by programmer */

DCL     1 A BASED(P),
          2 B FLOAT BINARY(24),
          2 C FLOAT BINARY(24);

/* Data definition supplied by CDD */

DCL     1 Q BASED,
          2 X BYTE_FIELD(8);
              .
              .
              .
P = ADDR(Q.X);
```

In this example, the ADDR built-in function gives the address of Q.X, which is assigned to P. Therefore, A can be used to reference X.

The following table summarizes the CDDL data types and corresponding PL/I data types. For further information on CDDL data types see the *VAX CDD/Plus*.

CDDL Data Type	PL/I Data Type	Remark
DATE	BYTE_FIELD(8)	
VIRTUAL	ignored	
BIT n ALIGNED	BIT(n) ALIGNED	
BIT n	BIT(n)	
UNSPECIFIED	BYTE_FIELD(n)	Depending on length
	BIT_FIELD(n)	of field
TEXT	CHARACTER(n)	
VARYING STRING	CHARACTER(n) VARYING	

D_FLOATING	FLOAT BINARY	Depending on BASE
	FLOAT DECIMAL	specified in CDDL
D_FLOATING COMPLEX	BYTE_FIELD(16)	
F_FLOATING	FLOAT BINARY	Depending on BASE
	FLOAT DECIMAL	specified in CDDL
F_FLOATING COMPLEX	BYTE_FIELD(8)	
G_FLOATING	FLOAT BINARY	Depending on BASE
	FLOAT DECIMAL	specified in CDDL
G_FLOATING COMPLEX	BYTE_FIELD(16)	
H_FLOATING	FLOAT BINARY	Depending on BASE
	FLOAT DECIMAL	specified in CDDL
H_FLOATING COMPLEX	BYTE_FIELD(32)	
SIGNED BYTE	FIXED BINARY(7)	
UNSIGNED BYTE	BYTE_FIELD(1)	
SIGNED WORD	FIXED BINARY(15)	
UNSIGNED WORD	BYTE_FIELD(2)	
SIGNED LONGWORD	FIXED BINARY(31)	
UNSIGNED LONGWORD	BYTE_FIELD(4)	
SIGNED QUADWORD	BYTE_FIELD(8)	
UNSIGNED QUADWORD	BYTE_FIELD(8)	
SIGNED OCTAWORD	BYTE_FIELD(16)	
UNSIGNED OCTAWORD	BYTE_FIELD(16)	
PACKED NUMERIC	FIXED DECIMAL	
SIGNED NUMERIC	BYTE_FIELD(n)	
UNSIGNED NUMERIC	PICTURE '(d)9V(s)9'	
LEFT OVERPUNCHED	PICTURE 'T(d)9V(s)9'	
LEFT SEPARATE	PICTURE 'S(d)9V(s)9'	
RIGHT OVERPUNCHED	PICTURE '(d)9V(s)9T'	
RIGHT SEPARATE	PICTURE '(d)9V(s)9S'	

PL/I ignores CDD features that are not supported by PL/I, but issues error messages when the features conflict with PL/I.

E.2 Creating CDD Structure Declarations

CDD source files must be written in the Common Data Dictionary Language (CDDL). You enter them into your file directory using a VMS editor utility, just as you would enter any other file into your directory. CDD source files should have a file type of DDL, which is the file type recognized by the CDD compiler.

Once you have created a CDD source file, you can invoke the CDDL compiler to insert your record definitions in the CDD. First, however, you can define CDDL as a global symbol by issuing the following command line or including it in your login command procedure:

```
$ CDDL:==$SYS$SYSTEM:CDDL
```

After you have done so, you need only enter CDDL to invoke the CDD compiler. For example, to compile your FILE.DDL and insert it into the CDD, enter the following:

```
$ CDDL FILE.DDL
```

The CDD compiler compiles the structure declaration and issues messages, if necessary. To correct errors in your structure declaration, invoke an editor and change the data definition text as needed, and then recompile. The CDD enters only the highest version of a file; if you attempt to compile another file with the same name, the CDD compiler issues an error message. However, the Dictionary Management Utility (DMU) permits updates. For further information see the *VAX CDD/Plus Utilities Reference Manual*.

E.3 Using the CDD

The %DICTIONARY statement incorporates VAX CDD data definitions into the current PL/I source file during compilation. The %DICTIONARY statement can occur anywhere in a PL/I source file. It has the following format:

```
%DICTIONARY cdd-path;
```

cdd-path
Is any preprocessor expression. The preprocessor expression is evaluated and converted to a character string, if necessary. The resulting character string is then interpreted as the full or relative path name of a CDD object. The resultant path name must conform to the rules for forming VAX CDD path names.

There are two types of CDD path name: full and relative. A *full* path name begins with CDD$TOP and specifies the given names of all its descendants; it is a complete path to the record definition. Descendant names are separated from each other by a period.

A *relative* path name begins with any generation name other than CDD$TOP, and specifies the given names of the descendants after that point. You can create a relative path by establishing a default directory with a logical name. For example:

```
$ DEFINE CDD$DEFAULT CDD$TOP.PLI
```

This logical name definition specifies the beginning of the CDD path name; thus, a relative path name specifies the remainder of the path to the record definition. Note also that a CDD path name beginning with CDD$TOP overrides the default CDD path name. For example, if you have a record with the following path name:

```
CDD$TOP.SALES.JONES.SALARY
```

You can specify a relative path name as follows:

```
%DICTIONARY 'SALARY';
```

Or you can specify an absolute path name as follows:

```
%DICTIONARY '_CDD$TOP.SALES.JONES.SALARY';
```

The compiler extracts the record definition from the CDD and inserts the PL/I structure declaration corresponding to the record description in the PL/I program.

A %DICTIONARY statement can appear as a statement by itself, or it can appear within a regular PL/I structure declaration. However, the resulting structures appear somewhat different, depending on the way %DICTIONARY is included.

If the %DICTIONARY statement is not embedded in a PL/I language statement (that is, if %DICTIONARY immediately follows a nonpreprocessor semicolon or is the first statement in the program), then the resulting structure is declared with the logical level 1, and the BASED storage attribute is furnished. The logical member levels increment from 2. For example:

```
DECLARE PRICE FIXED BINARY(31);
%DICTIONARY 'ACCOUNTS';
```

would result in a declaration of the form:

```
DECLARE PRICE FIXED BINARY(31);
DECLARE 1 ACCOUNTS BASED,
          2 NUMBER,
            3 LEDGER CHARACTER(3),
            3 SUBACCOUNT CHARACTER(5),
          2 DATE CHARACTER(12),
           .
           .
           .
```

Notice that in the previous example, ACCOUNTS is a relative dictionary path name.

If the %DICTIONARY statement is embedded in a PL/I language statement, as in a structure declaration, then the resulting structure is declared with no logical level and no storage attribute. Logical member numbers are supplied and incremented from 100. For example:

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          %DICTIONARY 'ACCOUNTS';   ,
          %DICTIONARY 'ADDRESSES';  ;
```

Notice the syntax in this example: the %DICTIONARY statement is terminated with the usual preprocessor terminator semicolon before the normal PL/I punctuation. The normal PL/I punctuation must also be included so that the final structure declaration will contain proper PL/I punctuation. At compile time, this declaration would result in a declaration of the following form:

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          100 ACCOUNTS,
            101 NUMBER,
               102 LEDGER CHARACTER(3),
               102 SUBACCOUNT CHARACTER (5),
            101 DATE CHARACTER(12),
            .
            .
            .
          100 ADDRESSES,
           .
           .
           .
```

When you extract a record definition from the CDD, you can choose to include this translated record in the program's listing by using the /LIST /SHOW=DICTIONARY qualifiers on the PLI command line.

CDD data definitions can contain explanatory text in the CDDL DESCRIPTION IS clause. If you specify /SHOW=DICTIONARY, this text is included in the PL/I listing comments. You can use these comments to indicate the data type of each structure and member. The punctuation for CDDL comments is the same as for other PL/I programs.

Even if you choose not to list the extracted record, the names, data types, and offsets of the CDD record definition are displayed in the program listing's allocation map.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z