

USER'S MANUAL · APRIL 1996

Endless PL/I for POSIX Systems

This manual gives an overview of the PL/I language and an explanation of PL/I program development. It describes the operation of the Endless PL/I for POSIX Systems compiler and the features of the operating system environment that are important to the PL/I programmer.

OPERATING SYSTEM AND VERSION

Linux

macOS

Tru64 UNIX

SOFTWARE VERSION

Endless PL/I for POSIX Systems Version 2.0

Endless Software Solutions

High Wycombe, Western Australia

ORDER NUMBER ESS-PFUUM-001

SECOND PRINTING (UPDATE), APRIL 1996 • ESS-PFUUM-001

Endless Software Solutions makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Endless Software Solutions or an authorized sublicensor.

No responsibility is assumed for the use or reliability of software on equipment that is not listed as supported in the Product Description.

Restricted Rights: Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Endless Software Solutions and Endless PL/I are trademarks of Endless Software Solutions.

Alpha AXP, AXP, CDD, DEC, DEC 4000, DECwindows, Digital, OpenVMS AXP, ULTRIX, VAX, OpenVMS, VT102, VT220, VT240, VT320, VT330, VT340, and the DIGITAL logo are trademarks of Digital Equipment Corporation.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. macOS is a trademark of Apple Inc. Tru64 is a trademark of Hewlett Packard Enterprise Development LP. POSIX is a registered trademark of the IEEE.

SAA and IBM are trademarks of International Business Machine Inc.

Stratus is a trademark of Stratus Computer Inc.

Endless PL/I for POSIX Systems includes the GNU readline and assembler software. See [Appendix A, GNU Software Information](#) for the entire text of the Free Software Foundation's GNU Copyleft.

Portions Copyright 1984-1990 FairCom Corporation. All Rights Reserved.

Endless Software Solutions requests your critical evaluation to assist in preparing future documentation. Please send any comments to support@es.solutions or by physical mail to:

Endless Software Solutions
PO Box 2128
HIGH WYCOMBE WA 6057

Copyright © 1995, 1996, 2026 Endless Software Solutions

Table of Contents

Preface	viii
1. Intended Audience	viii
2. Structure of This Manual	viii
3. Where to Find More Information	viii
4. Conventions	ix
1. Overview of Endless PL/I for POSIX Systems	1
2. Developing a PL/I Program	2
2.1. Creating Source Files	2
2.2. Compiling and Linking a PL/I Program	2
2.3. Running a PL/I Program	15
3. The File System	18
3.1. File Control	18
3.2. Using the UNIX File System for I/O	18
3.3. Error Handling	22
4. File Organizations and Input/Output	25
4.1. File Organizations	25
4.2. Access Modes for Record I/O	26
4.3. Record Formats	27
4.4. Stream Files	29
4.5. Relative Files	30
5. Options of the ENVIRONMENT Attribute	35
5.1. Specifying and Using Environment Options	35
5.2. Summary of ENVIRONMENT Options	37
5.3. APPEND Option	41
5.4. BLKSIZE Option	42
5.5. CONSECUTIVE Option	42
5.6. DEFAULT_FILE_NAME Option	43
5.7. DELETE Option	43
5.8. F, FB, FS, or FSB Option	44
5.9. FILE_ID Option	44
5.10. FILE_ID_TO Option	44
5.11. FIXED_LENGTH_RECORDS Option	45
5.12. GROUP_PROTECTION Option	45
5.13. MAXIMUM_RECORD_SIZE Option	46
5.14. OWNER_PROTECTION Option	47
5.15. RECORD_ID_ACCESS Option	48
5.16. RECSIZE Option	48
5.17. SCALARVARYING Option	49
5.18. SHARED_READ Option	50
5.19. SUPERSEDE Option	50
5.20. TEMPORARY Option	51
5.21. V, VB, VS, or VSB Option	52
5.22. WORLD_PROTECTION Option	52

5.23. ENVIRONMENT Options for File Protection and File Sharing	53
5.24. File Sharing	54
6. Input/Output Statement Options	55
6.1. Option Format	55
6.2. Summary of Input/Output Statement Options	55
7. File-Handling Built-In Subroutines	60
7.1. FLUSH Built-In Subroutine (DEC Dialect Only)	60
7.2. REWIND Built-In Subroutine (DEC Dialect Only)	60
8. Using Sort Subroutines (IBM Only)	62
8.1. General Information	62
8.2. Calling the Sort Subroutines	64
8.3. Input and Output Routines	69
8.4. Programming Examples	72
9. Error Handling	76
9.1. RESIGNAL Built-In Subroutine	76
9.2. VAXCONDITION Condition	76
9.3. ON-Unit Actions	78
9.4. Relationship of UNIX Signals to PL/I ON-Units	81
9.5. Search Path for ON-Units	85
9.6. Scope of ON-Units	90
9.7. ON-Unit Examples	90
9.8. Condition-Handling Built-In Functions	91
10. Using PL/I in the Common Language Environment	94
10.1. The UNIX Procedure Calling and Condition Handling Standard	94
10.2. Parameter-Passing Mechanisms	95
10.3. Calling Routines	100
A. GNU Software Information	102
A.1. GNU Copyleft	102
B. Run-Time Messages	108
B.1. Message Format	108
B.2. Messages with Identifications	109
B.3. Messages without Identifications	124
B.4. Indexed File Messages	127
B.5. Panic Messages	127
C. Compiler Messages	129
Index	171

List of Figures

4.1. Relative File	30
9.1. Resignaling a Condition	79
9.2. Unwinding the Call Stack	80
9.3. Execution of an ON-Unit	86
9.4. Search for an ON-Unit	88
9.5. Effect of Multiple Conditions	89

List of Tables

1. Documentation Conventions Table	ix
2.1. Alphabetic List of PL/I Driver Options	4
3.1. Default Process Stream Names	21
3.2. System File Names for Get and Put Statements	21
5.1. Description of Columns in Table 5.2, "Summary of DEC and ANSI Dialect ENVIRONMENT Options"	37
5.2. Summary of DEC and ANSI Dialect ENVIRONMENT Options	37
5.3. Summary of IBM Dialect ENVIRONMENT Options	38
5.4. ENVIRONMENT Options That Are Ignored	39
5.5. Default Record Sizes	47
5.6. Default Record Sizes	49
6.1. Summary of Input/Output Statement Options	55
8.1. Entry Points for the Sort Program	63
9.1. Corresponding Values	82
9.2. ONCODE Values for PL/I ON Conditions	92
10.1. Converted Data Types of Arguments Passed by Reference	96
10.2. Assignment of Data Type to Dummy Arguments	98
B.1. Indexed File Messages	127

List of Examples

4.1. Creating a Relative File 31

Preface

This manual describes how to use the Endless PL/I for POSIX Systems compiler on Linux, macOS, Tru64 UNIX and other POSIX systems, and contains detailed explanations of the extensions made to the standard PL/I language for this implementation. To aid in program development, it includes information on some UNIX commands and utilities. It also includes information to assist in writing PL/I programs that use features of the file system and the operating system.

1. Intended Audience

This manual is designed for programmers who have a working knowledge of PL/I and some familiarity with a POSIX operating system and the C shell (csh).

2. Structure of This Manual

This manual is divided into the following sections:

- [Chapter 1, *Overview of Endless PL/I for POSIX Systems*](#) gives an overview of the Endless PL/I for POSIX Systems compiler.
- [Chapter 2, *Developing a PL/I Program*](#) explains developing a PL/I program.
- [Chapter 3, *The File System*](#) describes the file system.
- [Chapter 4, *File Organizations and Input/Output*](#) explains record and stream input/output.
- [Chapter 5, *Options of the ENVIRONMENT Attribute*](#) summarizes ENVIRONMENT options.
- [Chapter 6, *Input/Output Statement Options*](#) explains input/output statement options.
- [Chapter 7, *File-Handling Built-In Subroutines*](#) explains file-handling built-in subroutines.
- [Chapter 8, *Using Sort Subroutines \(IBM Only\)*](#) describes the Sort program interface, valid for the ibm dialect.
- [Chapter 9, *Error Handling*](#) describes condition handling.
- [Chapter 10, *Using PL/I in the Common Language Environment*](#) explains using PL/I in the common language environment

[Appendix A, *GNU Software Information*](#) contains the Free Software Foundation's Copyleft.

[Appendix B, *Run-Time Messages*](#) describes the messages written by the run-time library.

[Appendix C, *Compiler Messages*](#) lists the diagnostic messages of the compiler.

3. Where to Find More Information

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

The Endless PL/I for POSIX Systems Installation Guide gives instructions on how to install the Endless PL/I for POSIX Systems compiler.

The manpages contain information on PL/I built-in functions and pseudovariables. For an introduction to manpages, type the following:

```
man 3pl1 intro Return
```

The documentation for your operating system gives complete information on it.

4. Conventions

For the purposes of this manual, the term PL/I refers to Endless PL/I for POSIX Systems, which runs on Linux, macOS, Tru64 UNIX and other POSIX systems.

All descriptions of the effects of executing statements and evaluating expressions assume that the initial procedure activation of the program is through an entry point with OPTIONS(MAIN).

It is further assumed that any non-PL/I procedures called by the program follow all conventions of the PL/I run-time environment.

Table 1. Documentation Conventions Table

Conventions	Meaning
Return	This symbol represents a single stroke of the RETURN key on the keyboard.
Ctrl+X	This symbol represents a control key combination. The letter X can be any keyboard character. To generate a control key combination, hold down the CTRL key while pressing the specified letter.
command	System text and code in examples and in text appears in the Courier typeface. Since the UNIX operating system differentiates between lowercase and uppercase characters, literal strings in examples and text must be entered exactly as shown.
bold	User input in interactive examples appears in the bold Courier typeface.
bold	Bold words in text indicate the first use of a new term.
italic	File and variable names appear in italics..
⋮	Vertical ellipses indicate that irrelevant parts of the program text or program output have been omitted..
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 (').
#	A # symbol is used in some contexts to indicate a single ASCII space character.)
p11(1)	Specifies a manual page. For more information, type: man section page
	This margin icon indicates that the paragraph contains important information to which you should pay close attention.
SYNTAX	Syntax (format) diagrams appear in the Helvetica typeface.
...	Horizontal ellipses indicate that additional parameters, options, or values can optionally be entered. When a comma precedes an ellipsis, it indicates that successive items must be separated by commas. Used in syntax diagrams.

Conventions	Meaning
[]	Square brackets indicate that a syntactic element is optional and you need not specify it. Used in syntax diagrams
[LIST] [EDIT]	Brackets surrounding two or more stacked items indicate conflicting options, one of which can optionally be chosen. Used in syntax diagrams
{ EXTERNAL } { INTERNAL }	Braces surrounding two or more stacked items indicate conflicting options, one of which must be chosen. Used in syntax diagrams
FILE (<i>file-reference</i>)	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 syntax diagrams, not to code examples.

Overview of Endless PL/I for POSIX Systems

Endless PL/I for POSIX Systems is a comprehensive and powerful programming language useful for systems programming, scientific computation, and commercial data handling and data organization. It is a block-structured language that lends itself to the creation of efficient and maintainable structured programs. It has extensive string-handling capabilities.

Endless PL/I for POSIX Systems is Endless Software Solutions' implementation, with extensions, of the American National Standard PL/I General-Purpose Subset, ANSI X3.74-1981. The General-Purpose Subset is a subset of full PL/I, 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 excludes features of PL/I that are prone to error, not often used, or that greatly increase the complexity of run-time support required by the compiler.

Over and above its conformity to the General-Purpose Subset, Endless PL/I for POSIX Systems offers many enhancements that extend the functionality of the 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, especially VAX PL/I. Other extensions integrate Endless PL/I for POSIX Systems into the common language environment of POSIX systems and into a complete programming environment. Endless PL/I for POSIX Systems extensions are intended for programs that will execute exclusively under the control of the UNIX operating system.

The Endless PL/I for POSIX Systems extensions provide support for the UNIX Procedure Calling and Condition Handling Standard, allowing PL/I procedures to call procedures written in other languages. 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.

Endless PL/I for POSIX Systems can be used with a number of tools in a complete programming environment. For example, the compiler system provides a source level, interactive debugger called dbg. This enables you to debug programs as they execute.

The Endless PL/I for POSIX Systems language is fully described in the Endless PL/I for POSIX Systems Reference Manual. This reference manual contains further information on the General-Purpose Subset, the UNIX extensions, and the differences between Endless PL/I for POSIX Systems and other implementations of the PL/I language.

Endless PL/I for POSIX Systems is compatible with other versions of Endless PL/I (Open VMS and RISC ULTRIX) with some exceptions. VMS specific options that are not compatible are emulated, flagged at compile-time, or ignored, depending on the VMS specific option. See the appendix on language differences in the Endless PL/I for POSIX Systems Reference Manual for more information.

Developing a PL/I Program

This chapter describes how to create, compile, link, and run a Endless PL/I for POSIX Systems program.

2.1. Creating Source Files

A PL/I source file is a UNIX file that contains PL/I code, as defined in the Endless PL/I for POSIX Systems Reference Manual.

To create a source file, use the text editor of your preference, such as the `vi` editor. You can also use any of the UNIX file management tools available on your system to generate source files. For example:

- You can obtain and incorporate any source files to which you have access from other file systems, or processors.
- Programs and shell scripts can generate PL/I code and write a source file for later compilation.

2.2. Compiling and Linking a PL/I Program

The following sections discuss the `p11` command and the command options.

2.2.1. The `p11` Command

A program called a driver invokes the major components of the compiler system. The components are:

- The PL/I front-end (`fp1i`)
- The assembler (`as`)
- The link editor (`ld`)

The `p11` command runs the driver that compiles, optimizes, assembles, and link edits your programs. The command may be followed immediately by one or more options, by one or more filenames, or both.

FORMAT

```
p11 [option] ... filename.suffix [filename.suffix ...]
```

option

A driver option. Driver options specify instructions to all the processing phases. You can specify either the long or short version of the options, as shown in [Table 2.1, “Alphabetic List of PL/I Driver Options”](#).

filename.suffix

An input source file containing the program or module to be compiled. You must specify a file suffix. Valid suffixes are:

- PL/I: `.pli` or `.p11`
- Object code: `.o`

You can include more than one file specification on the same command line by separating the file specifications with a space as follows:

```
p11 file1.pl1 file2.pl1
```

2.2.1.1. Compilation with p11

The driver command, `p11`, invokes the following processing phases to compile and optionally link edit the source file:

- PL/I compiler: The front-end processes the source file, producing an internal language called ucode. PL/I then converts the optimized ucode to assembly language.
- Assembly: Converts the assembly-language code to an object file.
- Link edit: Merges object files and libraries producing an executable binary.

By default, the `p11` command both compiles and link edits a source module. If you specify the `-c` driver option, processing stops after the assembler phase, producing a linkable object file. You can then use the `p11` or `ld` command to link edit the linkable object(s) into one executable binary. By default, the linkable object file has the same name as the source file, but with the suffix `.o`.

For example:

```
p11 file.pli -c
```

This produces the linkable object `file.o`.

The default name of the executable binary is `a.out`. For example:

```
p11 file.pl1
```

This command line produces the executable binary `a.out`.

You can specify a different name for the executable binary by using the driver option `-o` and supplying the desired file name. For example:

```
p11 file.pl1 -o file
```

This command line compiles and link edits the source module `file.pli` and produces an executable binary, `file`.

If you have previously compiled the source module and used the `-c` option to produce the linkable object `file.o`, the following command line links edit the object module:

```
p11 file.o -o file
```

This command link edits the object module `file.o` and produces the executable binary `file`. You can link edit several object modules together to produce a single executable binary, as follows:

```
p11 file1.o file2.o -o file
```

This command link edits the object modules `file1.o` and `file2.o` and produces the executable binary file.

When specifying a name for your executable binary or object module, note that the full file name (file name and extension) must differ from the name of the source file(s). In other words, when compiling `file.pli`, do not supply `file.pli` as the full file name of your executable binary or object module.

2.2.1.2. Compiling Multi-Language Programs

When your application has two or more source programs written in different languages, including C, Pascal, and Fortran, compile each program module separately with the appropriate driver and then link them in a separate step. To create the linkable objects, use the `p11` command option `-c`. The following example compiles a PL/I program and a C program:

```
p11 -c plifile.pli
cc -c cfile.c
```

When you compile the routines, the drivers produce the linkable object files `plifile.o` and `cfile.o`. Use the `p11` command to perform the link edit, which loads and initializes the appropriate run-time libraries for PL/I.

2.2.1.3. Driver Options

Table 2.1, “Alphabetic List of PL/I Driver Options” lists the options for the `p11` command line. The UNIX manpage `p11(1)` also describes the `p11` command options if the manual pages are installed on the system.

Table 2.1. Alphabetic List of PL/I Driver Options

Option	Long Option	Meaning
<code>-a</code>	<code>--align</code>	Specifies natural alignment. On Alpha processors, <code>-a</code> is equivalent to <code>-a8</code>
<code>-an</code>	<code>--align=n</code>	Aligns to the specified byte boundary. <code>n</code> must be a power of two. The default value is <code>-a1</code> , which causes the compiler to align so there are no pad bytes in structures. Note: Some programs rely on structures not being padded for alignment.
<code>-c</code>	<code>--compile_only</code>	Suppresses the loading phase of the compilation. Produces an object file even when you compile only one program.
<code>-D dialect</code>	<code>--dialect=diect</code>	Defines the dialect of the compiler, where <code>dialect</code> is one of the following: <code>ibm</code> , <code>dec</code> , <code>ansi</code> . The default dialect is <code>ansi</code> . See the Endless PL/I for POSIX Systems Reference Manual for information on the differences in dialects.
<code>-g</code>	<code>--debug</code>	Produces debugger symbol table information. This option implies the <code>-N</code> option.
<code>-Idir</code>	<code>--include_dir=dir</code>	Defines an additional directory for the PL/I preprocessor to search when looking for <code>%INCLUDE</code> modules You can specify this option multiple times to define several directories.
<code>-k</code>	<code>--check</code>	Enables range and subscript checking. Turns on signalling of the <code>STRINGRANGE</code> and <code>SUBSCRIPTRANGE</code> conditions. Use this option to generate code to dynamically check the values of array, character, and bit

Option	Long Option	Meaning
		string indexes. See the Endless PL/I for POSIX Systems Reference Manual for more information on the STRINGRANGE and SUBSCRIPTRANGE conditions.
-lname	--library=name	When linking, search an object library named libname.a, where name is a string.
-L	--library_dir	Changes the algorithm of searching for libx.a or libx.b so the default directories are not searched. Use this option when you only want to search the directories specified by the ld(1) option -Ldir.
-Ldir	--library_dir=dir	Changes the algorithm of searching for libx.a or libx.b so that ld searches the specified directory before the default library directories.
-m lower ^a	--case lower	Specifies that all identifiers are case insensitive and externals are converted to lower case. This is the default.
-m mix	--case mix	Specifies that all identifiers are case sensitive. Allows mixed case identifiers
-m upper	--case upper	Specifies that all identifiers are case insensitive and externals are converted to upper case.
-M lower -M mix -M upper	--include_case lower --include_case mix --include_case upper	Specifies the case of the include file names in a preprocessor %INCLUDE statement. The specified case is applied when the module name is built by the preprocessor. -M lower is the default, specifying that all file names are converted to lowercase. -M mix specifies case sensitivity, allowing mixed case file names. -M upper converts filenames to upper-case.
-N	--non_shared	Links the image without shared libraries. By default, images are linked with shared libraries. You must link without shared libraries to use the dbg debugger. Thus the -N option is implied by the -g option.
-o file	--output file	Generates an executable binary with the specified file name. If you do not use this option, the compiler names the executable binary a.out.
-O	--optimize	Performs all optimizations, including global register allocation. Use this option to create ucode object files for all PL/I source files. The newly created ucode object files, ucode object files specified on the command line, the run-time start-up routine, and all the run-time libraries are ucode linked. The compiler optimizes the resulting ucode linked file, then links it as usual, producing an executable binary. The ucode linking does not produce an .o file. Restrictions: The -c option cannot be specified with the -O option. The -O option must precede all source file arguments.
-p or -p1	--profile or --profile=1	Generate information for profiling. Use this option to prepare for profiling by periodically sampling the program counter value. The option affects run-time loading. When loading occurs, the standard run-time start-up is replaced by the profiling run-time start-up routine. The level 1 profiling library libprof1.a is searched.
-p0	--profile=0	Do not generate information for profiling. This is the default.

Option	Long Option	Meaning
-P or -P1	--preprocessor or --preprocessor=1	Turns on the preprocessor phase of compilation. This is the default for the -D dec dialect. See the Endless PL/I for POSIX Systems Reference Manual for more information on the preprocessor.
-P0	--preprocessor=0	Turns off the preprocessor phase of compilation. This is the default for the -D ibm and -D ansi dialects.
-s	--list	Generates a compiler listing file with corresponding filename and a suffix of .1. The listing file includes source program lines with all the included files expanded, and a datamap of the program.
-T variant	--variant variant	Defines the compilation variant used by the VARIANT preprocessor built-in function. Ignored if you compile without using the preprocessor. See the Endless PL/I for POSIX Systems Reference Manual for more information.
-u suffixname	--suffix suffixname	Defines the file suffix for include files. Supplied for compatibility with Digital and valid for the -D dec dialect only.
-v	--verbose	Displays the compiler passes, with their arguments, input, and output files, as they execute. Also displays resource usage in the C shell time format.
-w	--nowarn	Suppresses compiler warning messages.
-x	--xref	Includes cross reference information in the listing file. Implies -s; you need not use -s if you use -x.

^aIf you do not specify --case upper, do not declare an external symbol that will map to "main" (lowercase). This will conflict with the library function "main" supplied in the Endless PL/I for POSIX Systems run-time library. In addition, the file symbols "SYSPRINT", "SYSIN", and "SYSOUT" are always case insensitive regardless of the specified compiler option. In other words, whether you specify "sysprint", "Sysprint", or "SYSPRINT", you will always refer to the pre-declared PL/I files.

2.2.2. Linking Objects using the p11 Command

Use the p11 command to link edit separate objects into one executable binary when the main program is written in PL/I. The driver recognizes the .o suffix as the name of a file containing object code suitable for link editing and immediately invokes the link editor. For example:

```
p11 -o file mainfile1.o file2.o
```

Write the main program in PL/I and use the p11 command to link edit the program to insure correct initialization of the PL/I run-time library. This statement produces the executable binary file.

The p11 command implicitly calls the ld link editor (unless you specify the -c compile-only option). The link editor combines object modules to produce an executable binary. These object modules include:

- Object modules previously produced during the operation of p11
- Object modules in .o files that you name on the p11 command line
- The PL/I run-time library
- Object modules from specified libraries

A `p11` command where all the parameters are object files does only a link edit. For example, such a command might appear in a file for `make(1)` that specifies a build procedure. This file separates the compile operations from the link edit so that `make` can omit operations on source files that have not been modified.

If possible, use the `p11` command instead of `ld` to link edit PL/I object files. This automatically selects the correct run-time libraries and the correct version of `ld`. However, if you need to specify options to `ld`, you must invoke `ld` directly. Refer to the manpage on `ld(1)` for more information on specifying `ld` options, including specifying the run-time libraries.

2.2.2.1. Run-Time Libraries

The object code for the PL/I built-in routines is in an archive library. The distinctive suffix `.a` identifies a file as an archive library. The link editor gets your program code from your object file or files, and then gets the built-in routine code from object libraries called run-time libraries. It merges them and resolves references.

Endless PL/I for POSIX Systems includes the following run-time libraries:

- `libpli.a`

The default run-time library for the `p11` command. It contains support for all PL/I language features except indexed file support (files declared or opened using the `ENVIRONMENT(INDEXED)` option). Programs using indexed files that are linked to this library generate fatal run-time errors.

- `libplit.a`

Contains the same features as `libpli.a` with the addition of indexed file support (see `libpli.a`). Because the library increases the size of executable binaries, slows the link editing process, and requires additional memory during execution, use it only when indexed file support is required.

Note

Indexed file support is an add-on option to Endless PL/I for POSIX Systems.

- `libpliisam.a`

Contains additional support functions required by `libplit.a`.

To specify other libraries when invoking the link editor, use the `-l` option. Library names start with `lib` and have extensions `.so` (shared object), `.a` (archive) or `.b` (ucode). For example, if you have written a PL/I program that calls routines from the run-time library `libx11.a`, then specify:

```
p11 . . . -lx11
```

2.2.2.2. Library Search Path

The link editor (`ld`) searches for any libraries you specify using the default search path listed in the manpage for `ld`. However, the system administrator or installer determines the actual location of the PL/I run-time libraries, so you might need to add directories to the default search path.

The `-L` option to the link editor, when followed by a directory name, adds the specified directory to the search path for run-time libraries. Do not type any separators between `-L` and the name of the directory. You can

use this option several times; the link editor searches the directories in the order that you specify on the command line. Your definition of the search path must precede your specification of libraries (using the `-l` option described in [Section 2.2.2.1, “Run-Time Libraries”](#)) on the command line.

The link editor searches the directories you specify with `-L`, in the order you specify them on the command line, before it searches the standard directories (which were specified during installation). This lets you specify files that supersede the usual run-time library.

If you use the `-L` option and do not follow it with a directory name, the link editor does not search its standard directories.

When you use the `-L` option, the link editor uses the following search path:

1. 1st directory specified by `-L`
2. 2nd directory specified by `-L`
3. Last directory specified by `-L`
4. The default search directories, as listed in the manpage for `ld` on your system, unless you specified the `-L` option without a path

2.2.3. PL/I Preprocessor

The PL/I preprocessor lets you change a source program at compile time. You can mix preprocessor statements with nonpreprocessor statements in the source program, but the preprocessor statements only execute at compile time. The resulting source program is then used for further compilation.

Note

By default, the preprocessor is turned on if you specify the `-D dec` command line option, and off if you specify either the `-D ibm` or `-D ansi` command line option, since the preprocessor provides compatibility with Digital implementations of PL/I.

The preprocessor does two types of preprocessing:

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

Preprocessor statements let you include text from other sources (`%INCLUDE` statements), control the course of compilation (`%DO`, `%GOTO`, `%VARIANT`, `%PROCEDURE`, and `%IF`), issue user-generated diagnostic messages, and selectively control listings and formats. The preprocessor statements are described in full in the Endless PL/I for POSIX Systems Reference Manual.

2.2.3.1. 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 binary 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(60) VARYING INITIAL(T);

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

This example illustrates several aspects of the preprocessor. First, you must compile this program 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 you only use the value of T 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 off of string constants. To ensure that the run-time program also has quotes around the string, you must supply additional quotes, as shown.

2.2.3.2. 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 (using 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 using 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).

You can use either of the following two types of argument lists in preprocessor procedures:

- Positional argument lists end with a right parenthesis. They use parameters sequentially, as in a parenthesized list. Positional argument lists can be used in any preprocessor procedure.
- Keyword argument lists end with a semicolon. They 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. You can only use keyword argument lists when the preprocessor procedure contains the STATEMENT option and is invoked from a nonpreprocessor statement.

Since a keyword argument list ends with a semicolon rather than a right parenthesis, the STATEMENT option lets you use a preprocessor procedure that has a keyword argument list as if it were a statement. Consequently, preprocessor procedures using the STATEMENT option let you extend the PL/I language by simulating otherwise unavailable features.

PREPROCESSOR STATEMENTS AND PROCEDURES

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, and must be the first item, other than white space (spaces or tabs) on the line. Preprocessor built-in functions are contained within preprocessor statements and consequently do not require a percent sign.

You can label preprocessor statements. Like other PL/I labels, you can use labels on preprocessor statements as the targets of program control statements. A preprocessor label must be an unsubscripted label constant preceded by a percent sign. The format for a preprocessor label is:

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

As with other preprocessor statements, the percent sign alerts the compiler that all text until the semicolon line terminator is preprocessor text. Therefore, you do not need to enter another percent sign on that line.

The compiler treats all statements within a preprocessor procedure as preprocessor statements. You must label preprocessor procedures to invoke them, and you must use a percent sign before the procedure label. However, statements within the procedure do not require leading percent signs.

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

2.2.4. Compiler Listing Files

If you specify either the `-s` or `-x` compiler option, the PL/I compiler produces a listing file. The listing file contains the following information:

- Source program lines
- Expansions of all included files
- A datamap
- (with the `-x` option) A cross reference listing
- error messages

The following example shows portions of a listing file:

```
source file: /lang/tmp/real_progAAAaagzEa.pl ❶
compiled on: 950923 at: 17:24:54 by: ANSI/IBM PL/I, rev 1.04in: /local/home/louise/test
Listing File is in: /local/home/louise/test/

    0-1    0      0      Test: procedure options (main); ❷
    0-2    1      0
    0-3    1      0      declare x character (8) initial ('foo'),
    0-4    1      0                      input_x character (8) initial ('');
    0-5    1      0      declare y fixed bin (15) initial (42),
    0-6    1      0                      input_y fixed bin (15) initial (0);
    0-7    1      0      declare valid bit(1) initial ('0'B);
    0-8    1      0
    0-9    1      0      put skip list ('What is the answer?: ');
    0-10   1      0      get list (input_y);
```

```

0-11  1      0      put skip list ('Type the most common file name: ');
0-12  1      0      get list (input_x);
0-13  1      0
0-14  1      0      if (x = input_x) & (y = input_y)
0-15  1      0      then
0-16  1      0          valid=real_programmer();
0-17  1      0      else begin;
0-18  2      0          put skip list ('You're not a real programmer!');
0-19  2      0          goto exit;
0-20  2      0      end;
0-21  1      0
0-22  1      0      real_programmer: procedure() returns(bit(1));
0-23  2      0
0-24  2      0      /* this exists to show another subroutine level */
0-25 * 2      0          /* in the listing file */
0-26  2      0
0-27  2      0      declare real_prog bit(1);
0-28  2      0
0-29  2      0      put skip list ('You're a Real Programmer!!!');
0-30  2      0      real_prog = '1'B;
0-31  2      0
0-32  2      0      end; /* real_programmer */
0-33  1      0
0-34  1      0      exit:
0-35  1      0
0-36  1      0      end Test;

```

<FF>

***** EXTERNAL ENTRY POINTS ***** ③

name	class	size	loc	attributes
------	-------	------	-----	------------

TEST	constant			ENTRY EXTERNAL
------	----------	--	--	----------------

procedure TEST on line 1

name	class	size	loc	attributes
------	-------	------	-----	------------

EXIT	constant			LABEL
------	----------	--	--	-------

def	ref	set
-----	-----	-----

34

19

INPUT_X	automatic		8c	84	CHAR(8) UNALIGNED INITIAL
---------	-----------	--	----	----	---------------------------

def	ref	set
-----	-----	-----

4

12

14

INPUT_Y	automatic		1	96	FIXED BIN(15,0) ALIGNED PRECISION INITIAL
---------	-----------	--	---	----	---

def	ref	set
-----	-----	-----

6

10

14

```

REAL_PROGRAMMER
    constant
        def      ref      set      ENTRY RETURNS INTERNAL
        22
        16

SYSIN  constant
        def      ref      set      FILE EXTERNAL
        1
        10
        12

SYSPRINT  constant
        def      ref      set      FILE EXTERNAL
        1
        9
        11
        18
        29

VALID  automatic      1b      112      BIT(1) UNALIGNED INITIAL
        def      ref      set
        7
        16

X      automatic      8c      92      CHAR(8) UNALIGNED INITIAL
        def      ref      set
        3
        14

Y      automatic      1      128      FIXED BIN(15,0) ALIGNED PRECISION INITIAL
        def      ref      set
        5
        14

begin block on line 17

no declared names

procedure REAL_PROGRAMMER on line 22

name    class    size    loc    attributes

REAL_PROG      automatic      1b      96      BIT(1) UNALIGNED
        def      ref      set
        27
        30

***** ENTRY POINTS AND ARRAY OF LABELS *****

name    class    size    loc    attributes

TEST  constant
REAL_PROGRAMMER      ENTRY EXTERNAL

```

```

      constant
      def      ref      set
      22
      16

<FF>

*****ERROR 179 SEVERITY 1 on line 1 in file ④ /lang/tmp/real_progAAAagzEa.pl.*****
The undeclared name "SYSIN" has been contextually declared
in this block.  It will acquire default attributes.

*****ERROR 179 SEVERITY 1 on line 1 in file /lang/tmp/real_progAAAagzEa.pl.*****
The undeclared name "SYSPRINT" has been contextually
declared in this block.  It will acquire default attributes.

```

The following list describes the callouts in the preceding listing file example:

- ❶ Header information—contains information about the source file you compiled, such as when you compiled it, the compiler variant used (dec, ansi, ibm), and the directory specification of the listing file.
- ❷ The source program listing—contains line numbers generated by the compiler, an optional asterisk (*) if the line is continued from a previous line, a level number, specifying the level of procedural nesting, a second level number, specifying the block level within a procedure, and the source text.
- ❸ Cross reference listing tables—includes tables listing external entry points, then a block-by-block listing of each procedure or internal block in the program, describing each variable declared in the block.
- ❹ Compilation error messages—the text of any error messages generated during compilation.

2.2.5. Compiler Error Messages

The PL/I compiler identifies syntax errors and violations of language rules in the source program. If the compiler locates any errors, it writes messages to standard output. If you enter the `p11` command interactively, the messages are displayed on your screen.

If the compiler creates a listing file it also writes these messages to the listing, as shown in the previous section.

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

```
*****ERROR nn SEVERITY s on line line-no in file filename *****
message-text
```

The `nn` specifies the error number.

The `s` specifies the severity of the error. The following letters represent possible severities:

- 4—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.
- 3—Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.

- 2—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.

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.
- 1—Informational. 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. You may, however, want to check that your code performs as expected, and possibly modify the specified line.

The filename indicates the file specification.

The 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. The error message also appears in the listing file (if listing is requested) with the listing line number.

The *message-text* is the 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.

For a message of severity 1, the first word is WARNING instead of ERROR. [Appendix C, Compiler Messages](#) lists the messages by number, with an explanation of each and the action to take.

2.3. Running a PL/I Program

To run a PL/I program, enter the name of the executable binary produced by the `p11` command. If you did not use the `-o` option in the `p11` command, the default name of the executable binary is `a.out`. Enter `a.out` to run your PL/I program. If you used the `-o` option and specified a filename, enter the filename to run your program.

2.3.1. Run-Time Errors

If an error occurs at run-time and the program does not handle it, the run-time library writes a report of it on standard error and ends the program. See [Appendix B, Run-Time Messages](#) for the format of the report and a description of the messages.

2.3.2. Passing Command Line Parameters to a PL/I Program

You can pass command line parameters to your PL/I program, and access the parameters in your program. The following sections show how to write your code to access command line parameters, and how to run a program and specify command line parameters.

2.3.2.1. Accessing Command Line Parameters in PL/I Programs

Endless PL/I for POSIX Systems provides two methods to access command line parameters:

1. Through procedure parameters in the main program.

In your main program, you can declare up to ten parameters to be passed from the command line when users run your program. You must declare these parameters CHARACTER VARYING, giving the actual length of the string as the maximum length.

For example:

```
PARAM: PROCEDURE (name,address,city,state,zip)
          OPTIONS(MAIN);
DECLARE (name,address,city,state,zip) CHAR(100) VARYING;
```

Your program can access these parameters only if the parameters are actually passed upon program invocation. To check the number of specified parameters, reference the external variable ARGC (argument count), included in the global definitions file `p11std.in`, as shown in the example below.

1. Through array of pointers to parameters.

You can access the command line parameters using the global variable ARGV (argument pointer), a pointer to an array of pointers to the actual parameters passed to the program. Again you can reference only as many parameters as were actually passed on the command line.

Both ARGC and ARGV are included in the global declarations include file, `p11std.in`, supplied with Endless PL/I for POSIX Systems.

The following example shows both methods of accessing command line parameters:

```
%INCLUDE p11std;

PARAM_TEST: PROCEDURE (name, address, city, state, zip)          OPTIONS(MAIN);
DECLARE (name, address, city, state, zip) CHARACTER(100)        VARYING;
DECLARE i FIXED BIN;

IF ARGC >= 1 THEN
    PUT SKIP EDIT ('Name: ',Name) (a,a);
ELSE
    PUT SKIP EDIT ('Name is missing') (a);

IF ARGC >= 2 THEN
    PUT SKIP EDIT ('Address: ',Address) (a,a);
ELSE
    PUT SKIP EDIT ('Address is missing') (a);

IF ARGC >= 3 THEN
    PUT SKIP EDIT ('City: ',City) (a,a);
ELSE
    PUT SKIP EDIT ('City is missing') (a);
```

```
IF ARGC >= 4 THEN
    PUT SKIP EDIT ('State: ',State) (a,a);
ELSE
    PUT SKIP EDIT ('State is missing') (a);

IF ARGC >= 5 THEN
    PUT SKIP EDIT ('Zip: ',Zip) (a,a);
ELSE
    PUT SKIP EDIT ('Zip is missing') (a);

DO I = 1 TO ARGC;
    PUT SKIP EDIT ('Parameter ',i, 'is ',ARGS(i)->ARG_CV) (a,f(2),a,a);
END; /* of do loop */

END; /* of program */
```

2.3.2.2. Specifying Command Line Parameters

To specify command line parameters, type them after the command that runs your PL/I program. For example, if you compile the preceding example using the `-o` command line option, and call your program `param_test`, run it as follows:

```
% param_test "Jocelyn Wood" "92 Forest Path" "Groveland" "New Hampshire" "01110"
```

Note

Command line format varies in different UNIX shells. If this format does not work, see the manual pages for your shell for the correct format.

The File System

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

- Declaring, opening, and closing files
- The relationship between the I/O statements in PL/I and the I/O procedures of the operating system
- POSIX file-naming and file-defining conventions
- File system error handling at run time

Chapter 4, *File Organizations and Input/Output*, Chapter 5, *Options of the ENVIRONMENT Attribute*, Chapter 6, *Input/Output Statement Options*, and Chapter 7, *File-Handling Built-In Subroutines* give additional information on the file system, such as stream and record I/O, options (ENVIRONMENT and I/O statement options), and built-in subroutines for file handling.

3.1. 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 that describe 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 that refers 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 Chapter 4, *File Organizations and Input/Output*, "Chapter 4, *File Organizations and Input/Output*".

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 on the CLOSE statement; the ENVIRONMENT options are summarized in Chapter 5, *Options of the ENVIRONMENT Attribute*.

Declaring, opening, and closing files are discussed in more detail in the Endless PL/I for POSIX Systems Reference Manual.

3.2. Using the UNIX 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 PL/I run-time library routine. These routines in turn make calls to the appropriate operating system services.

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

- Options of the ENVIRONMENT attribute
- Keyword options on I/O statements in PL/I
- Built-in subroutines

3.2.1. PL/I Files and UNIX 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.

By default, 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 a UNIX file. The default file would be F. To change the name, use the title option of the DEFAULT_FILE_NAME environment option. Refer to [Section 5.6, “DEFAULT_FILE_NAME Option”](#) for more information.

The following sections describe in more detail how PL/I associates a file constant with a device or file.

3.2.2. Using the TITLE Option

When you specify the TITLE option on an OPEN statement, you can include all or part of a POSIX 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.

This file specification completely defines a file on the local system. When accessing the file OUTFILE, the program will be written to the file /usr/ payroll/ january.

```
OPEN FILE (OUTFILE)
TITLE ('/usr/payroll/january');
```

The following is an example of using a variable to specify the title:

```
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.

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.

If the value of the specified title begins with a dollar sign (\$) the value is interpreted as an environment variable and substituted accordingly. If the substituted value contains a dollar sign (\$) it is also interpreted as an environment variable. This process terminates when no further substitutions can be made or there are no more dollar signs.

Upon completion of the translation phase, if the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute is used, the name associated with the file is merged with the DEFAULT_FILE_NAME option.

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. Note that the final file specification cannot exceed the value of MAXPATHLEN as defined in <sys/param.h>.

3.2.3. Using Environment Variables

Before executing a program at the shell level, you can create an environment variable to assign a UNIX 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 OUTPUT;
```

The local file specification created on the local system is INFILE.

3.2.3.1. Environment Variables in TITLE Values

The value specified in a TITLE option can represent an environment variable when it starts with a dollar sign (\$) For example:

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

If, before running the program, you define an environment variable called \$NEWFILE, the OPEN statement would use the value of NEWFILE as the title. For example:

```
% setenv NEWFILE /usr/users/you/databases/phonebook
```

(This command may differ on your shell.) The file opened would be:

```
/usr/users/you/databases/phonebook
```

3.2.3.2. Process Standard System File Names

The system provides every process with a default set of standard name assignments, as listed in [Table 3.1](#), “Default Process Stream Names”. Because the files associated with these assignments exist for the life of the process, and because they are permanently open, they are called *process permanent files*.

Table 3.1. Default Process Stream Names

Stream Name	Default Equivalence Name
stdin	Standard input. For an interactive user, this is the terminal or, if redirected, the input command file.
stdout	Standard output. For an interactive user, this is the terminal or, if redirected, the output file name specified.

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

Table 3.2. System File Names for Get and Put Statements

Statement	Default PL/I File	Default Title
GET	SYSIN	stdin
PUT	SYSPRINT	stdout

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 reads from standard input.

A similar set of associations occurs when a program executes a PUT statement without the FILE option: the resulting output is written to standard output, stdout.

In the dec and ansi dialects, the process permanent file SYSOUT is also mapped to stdout.

The file symbols "SYSPRINT", "SYSIN", and "SYSOUT" are always case insensitive regardless of the specified compiler -m or --case option. In other words, whether you specify "sysprint", "Sysprint", or "SYSPRINT", you will always refer to the pre-declared PL/I files.

The file symbols "SYS\$PRINT", "SYS\$INPUT", and "SYS\$OUTPUT" are not predeclared. However, if you declare any of these, they map to the same files as the predeclared process permanent files, "SYSPRINT", "SYSIN", and "SYSOUT".

3.2.4. Expanding File Specifications

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

```
STATE_NAME = NEBRASKA
```

```
STATES = /usr/BACKUP
```

They also assume the following current default device and directory:

```
/usr/users/malcolm
```

Example 1:

```
DECLARE STATES FILE RECORD OUTPUT;
OPEN FILE(STATES) TITLE ('/usr/STATE.DATA/NEVADA');
```

Steps:

1. Obtain the value in the TITLE option. This value represents the complete file specification.

Final Specification: /usr/STATE.DATA/NEVADA

Example 2:

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

Steps:

1. Obtain the value from the TITLE option, \$STATE_NAME.
2. Translate the environment variable STATE_NAME to obtain the equivalence, NEBRASKA.

Final Specification: /usr/users/malcolm/NEBRASKA

Example 3:

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

Steps:

1. Apply the default title, STATES.
2. Translate the environment variable STATES to obtain the equivalence name, /usr/BACKUP.

Final Specification: /usr/BACKUP

3.3. Error Handling

Endless PL/I for POSIX Systems uses 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.

In the IBM dialect only, the following ON conditions are also used:

- The NAME condition handles errors involving data-directed GET statements with the FILE option.
- The RECORD condition handles size errors involving records during READ, WRITE, or REWRITE operations.
- The TRANSMIT condition handles errors involving uncorrectable transmission errors during record I/O.

See the Endless PL/I for POSIX Systems Reference Manual for more information on these conditions.

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 ('input');
```

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

3.3.1. 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 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 file system or from the PL/I run-time system. Note that file system errors are mapped to RMS (VAX Record Management Services) errors by the PL/I run-time library for compatibility with VAX PL/I.

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.

3.3.2. 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 the system include file \$RMSDEF (/usr/lib/cmplrs/pli100/usr/include/rmsdef.in).

3.3.3. 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 UNIX 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:

```

PL/I unhandled exception at 0x00401380:
PL/I ERROR condition.
PL/I UNDEFINEDFILE condition on file 'INFILE'.
FILENAME: '/usr/users/tim/databases/myfile.dat'.
FILENAME, FNF, File not found.

```

In this example, the error occurred because an OPEN statement could not find the specified file. This fatal error terminates the program.

File Organizations and 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—stream and relative
- Access modes—sequential, random, and random/sequential, and record identification
- Record formats—variable-length, fixed-length
- Stream files
- Relative files

For position information and descriptions of the READ, WRITE, DELETE, and REWRITE statements, see the Endless PL/I for POSIX Systems Reference Manual.

4.1. File Organizations

The UNIX file system supports only stream files; however, the PL/I run-time system simulates the behavior of relative files by imposing restrictions on the access of files having these organizations.

One of the consequences of this is that (unlike VAX PL/I) the Endless PL/I for POSIX Systems run-time system relies on the file organization and access method specified in the DECLARE, OPEN, and READ statements of your PL/I program to determine the file organization. Thus it is up to the programmer to ensure that the correct file organization is specified in the input/output statements of the PL/I program; otherwise, results are unpredictable. To avoid problems when specifying the file, always open a file with the same file attributes.

RESTRICTIONS

Endless PL/I for POSIX Systems does not support some file organizations or combinations of file organizations and access methods due to limitations of the file system. These are:

- Stream access of relative file with fixed or variable length records
- Sequential access of stream file with variable length records (supported by VAX PL/I)
- Direct access of stream file with variable length records (supported by VAX PL/I)

The following file organizations are supported by PL/I:

- Stream
- Relative

STREAM FILES

In a stream file, characters are ordered one after the other, in the order in which they are written. New characters can be added only at the end of an existing file. Stream files are discussed in more detail in [Section 4.4, “Stream Files”](#).

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 4.5, “Relative Files”](#).

4.2. Access Modes for Record I/O

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, there is an additional way to access a file. Specify the ENVIRONMENT option RECORD_ID_ACCESS to indicate 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.

4.2.1. Sequential Access

You can use sequential access with any type of file organization except stream access to a variable length record file. 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.

4.2.2. Direct Access

When you access a file randomly by key, each I/O request must contain the KEY or KEYFROM option. For a relative file, the key is the relative record number of the record to be accessed.

By default, a READ statement accesses a record based on an exact match of the key specified. In PL/I, 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.

4.2.3. Direct 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.

4.2.4. Access by Record Identification

You can use a record identification to access records in any type of disk file (except stream files with variable length records) that is opened for any type of access. To access records by record identification:

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.

4.3. Record Formats

The following types of record formats are allowed for all file organizations:

- Fixed length
- Variable length

You must specify the format when you create a file and each time you open the file. This is necessary because PL/I cannot determine the format of the records in the file.

4.3.1. 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 after the file is created.

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.

4.3.1.1. IBM Dialect Fixed Length Environment Options

The following environment options are available in the IBM dialect only:

- `F`, `FB`, `FS`, or `FBS`—all equivalent to `FIXED_LENGTH_RECORDS`
- `RECSIZE`—equivalent to `MAXIMUM_RECORD_SIZE`

4.3.2. Variable-Length Records

PL/I supports only record I/O on variable-length record files with relative file organizations. In a file containing variable-length records, each record can have a different size. PL/I 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 files. Use 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.

4.3.2.1. IBM Dialect Variable Length Environment Options

The following environment options are available in the IBM dialect only:

- V, VB, VS, or VBS—all equivalent to the variable length record format
- RECSIZE—equivalent to MAXIMUM_RECORD_SIZE

4.4. Stream Files

Record I/O on stream files is limited to those with fixed-length records. This section shows examples of some typical sequential file I/O operations on stream disk files.

4.4.1. Stream Files Organization

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

A file has the STREAM attribute if it meets any of the following conditions:

- 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 declared with the STREAM attribute have the following characteristics:

- They are a stream of bytes.
- They have variable length records. The records are delimited by the newline character (\n).

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.

Unlike VAX VMS, an existing stream file cannot be reopened and accessed by the READ and WRITE statements. Stream files are true byte streams on UNIX. If READ and WRITE open a file implicitly, the attributes RECORD INPUT SEQUENTIAL and RECORD OUTPUT SEQUENTIAL are implied, respectively. These attributes are not compatible with an existing disk file created with the STREAM and OUTPUT attributes, and these operations will produce unpredictable results.

4.4.2. Creating a Stream File

Whenever a PL/I program opens a file with the SEQUENTIAL OUTPUT attributes, PL/I creates a new stream file. By default, records are 510-byte, variable-length records. If record I/O is done on this file, you must specify the FIXED_LENGTH_RECORDS attribute (or the IBM dialect equivalent attribute).¹

4.4.2.1. Appending Records to an Existing File

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

¹The default is preserved for compatibility with VAX PL/I. Endless PL/I for POSIX Systems does not support SEQUENTIAL (record) access to variable length stream files.

```

OPEN FILE (BIRD_FILE) OUTPUT SEQUENTIAL
  TITLE ('BIRDS.DAT')
  ENVIRONMENT (APPEND, FIXED_LENGTH_RECORDS);
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.

4.4.2.2. Superseding an Existing File

The ENVIRONMENT option SUPERSEDE lets you create a new version of a file each time you open it, simultaneously deleting an existing version. This is the default behavior for output files in PL/I. For example, the following OPEN statement opens the file CONTROL.DAT, deleting any pre-existing versions of the file.

```

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

```

4.5. Relative Files

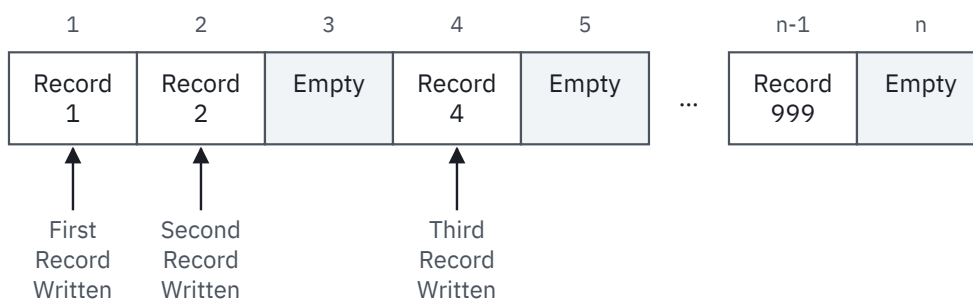
This section describes the organization of a relative file, suggests considerations for creating and using relative files, and shows examples of some typical relative file I/O operations.

4.5.1. 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 accessed.

Figure 4.1, “Relative File” 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 4.1. 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.

4.5.2. Creating a Relative File

In PL/I, 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 creates a relative file.

When you initially create a relative file in a PL/I program, consider using the ENVIRONMENT option MAXIMUM_RECORD_SIZE (or the IBM dialect RECSIZE option) to maximize the efficiency of I/O operations on the file.

MAXIMUM RECORD SIZE

When you specify the length of the records in a file, the value you specify is used in the MAXIMUM_RECORD_SIZE option (or the IBM dialect RECSIZE option) to calculate a cell size. It uses the following formulas to calculate the size:

- For all records—cell size = 2 + MAXIMUM_RECORD_SIZE (two bytes are required for overhead; they contain the 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)

When you select a record size for a relative file, 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 uses the default length of 480 bytes.

4.5.3. Using Relative Files

You can create a relative file from any existing file that is suitable for relative file organization. [Example 4.1](#), “Creating a Relative File” illustrates copying a sequential file with fixed-length records into a relative file.

Example 4.1. Creating a Relative File

```
COPY_TO_RELATIVE: PROCEDURE OPTIONS(MAIN);

    %INCLUDE PARTLIST; ❶ /* Declaration of PARTLIST */
    DECLARE OLDFILE FILE INPUT RECORD SEQUENTIAL ENVIRONMENT(
        FIXED_LENGTH_RECORDS,
        MAXIMUM_RECORD_SIZE(36));
    DECLARE RECORD_NUMBER FIXED BINARY(15);
    DECLARE
        PARTS FILE OUTPUT KEYED RECORD ENVIRONMENT(
            FIXED_LENGTH_RECORDS, ❷
            MAXIMUM_RECORD_SIZE(36));

    ON ENDFILE(OLDFILE) BEGIN; ❸
        CLOSE FILE(OLDFILE), FILE(PARTS);
        STOP;
    END;
```

```

OPEN FILE(OLDFILE), FILE(PARTS);

DO WHILE('1'B);
    READ FILE(OLDFILE) INTO(PARTLIST); ❹
    RECORD_NUMBER = PARTLIST.NUMBER;
    WRITE FILE(PARTS) FROM(PARTLIST) KEYFROM(RECORD_NUMBER); ❺
END;
END COPY_TO_RELATIVE;

```

The following notes are keyed to [Example 4.1, “Creating a Relative File”](#):

- ❶ 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.
- ❷ The relative file PARTS is declared with fixed-length, 36-byte records.
- ❸ 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.
- ❹ 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.
- ❺ 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.

4.5.3.1. Populating a Relative File

In [Example 4.1, “Creating a Relative File”](#), 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 specified for a record is 200, only enough space for 200 records is allocated.

4.5.3.2. 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 PART.LIST */
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.

4.5.3.3. 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.

4.5.4. 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 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.

```

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;
END;
GET_DATA:

```

In this example, the ON-unit gets symbolic names for a specific status value returned by ONCODE, from RMSDEF, a system include file provided for compatibility with VAX PL/I. The value RMS\$_REX indicates that a record already exists.

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 the system include file `rmsdef.in`, from `/usr/lib/pl1/include`.

Options of the ENVIRONMENT Attribute

The options of the ENVIRONMENT attribute provided by Endless PL/I for POSIX Systems let you do the following:

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

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 alphabetic order. The chapter concludes with a discussion of the environment options designed for file protection, file sharing, and I/O optimization.

5.1. Specifying and Using Environment Options

All environment options can be specified in the declaration of a file constant or, in the dec and ansi dialects, in an OPEN statement . Certain options can also be specified in a CLOSE statement.

If you declare a file constant or file variable explicitly or implicitly as EXTERNAL, you must use identical attributes, including ENVIRONMENT options, in all blocks that declare the file.

5.1.1. 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
- 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 to enable or disable 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.

5.1.1.1. 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(31).

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.

5.1.1.2. 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.

5.1.1.3. 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.

5.1.2. 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, they determine the option for the file if the OPEN option creates the file. Because the options are not stored with the file, they must be given for each opening of a file. The results are unpredictable if the options given do not match with those existing for the file.

5.1.3. 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 MAXIMUM_RECORD_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.

5.2. Summary of ENVIRONMENT Options

The options to the PL/I ENVIRONMENT attribute for the DEC and ANSI dialects are summarized in alphabetic order in Table 5.2, “Summary of DEC and ANSI Dialect ENVIRONMENT Options”, and the ENVIRONMENT options for the IBM dialect in Table 5.3, “Summary of IBM Dialect ENVIRONMENT Options”. Table 5.1, “Description of Columns in Table 5.2, “Summary of DEC and ANSI Dialect ENVIRONMENT Options”” describes the columns in Table 5.2, “Summary of DEC and ANSI Dialect ENVIRONMENT Options”.

The sections following these tables describe the Endless PL/I for POSIX Systems environment options in detail.

Table 5.1. Description of Columns in Table 5.2, “Summary of DEC and ANSI Dialect ENVIRONMENT Options”

Option	Gives the name of the ENVIRONMENT option and its argument, if any. An option that does not show an argument can be specified with a Boolean argument.
Usage	Briefly describes the option.
Specify At	Indicates when the option is meaningful. The possible items in this column are: Create—the option can be specified on a DECLARE or OPEN. It is meaningful 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.

Table 5.2. Summary of DEC and ANSI Dialect ENVIRONMENT Options

Option	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)
DEFAULT_FILE_NAME (expression)	Defines a default specification for a file.	Create Open	Record Stream	None	CHAR(128) VARYING

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
DELETE	Specifies that the file be deleted when it is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
FILE_ID (variable)	Identifies a file by its internal file identification.	Open	Record Stream	Not applicable	(6)FIXED BINARY(31)
FILE_ID_TO (variable)	Identifies a file by its internal file identification.	Create Open	Record Stream	Not applicable	(6)FIXED BINARY(31)
FIXED_LENGTH_RECORDS	Specifies a file with fixed-length records of a maximum record size.	Create	Record	Disabled	BIT(1)
GROUP_PROTECTION (expression)	Defines the type of file access allowed to members of the owner's group.	Create	Record Stream	Current process default ^a	CHAR(4)
MAXIMUM_RECORD_SIZE (expression)	Specifies the maximum size valid for records in the file.	Create	Record	Varies ^b	FIXED BINARY(31)
OWNER_PROTECTION (expression)	Specifies the type of file access allowed the owner of the file.	Create	Record Stream	Current process default ^a	CHAR(4) VARYING
RECORD_ID_ACCESS	Indicates that records will be accessed by internal file system identification.	Create Open	Record	Disabled	BIT(1)
SHARED_READ	Lets other users read records in the file.	Create Open	Record	Enabled ^c	BIT(1)
TEMPORARY	Specifies a temporary file which is deleted when the process terminates.	Create	Record Stream	Disabled	BIT(1)
WORLD_PROTECTION (expression)	Specifies the type of file access allowed to general system users.	Create	Record Stream	Current process default ???	CHAR(4) VARYING

^aSee umask(1)^b512 bytes for sequential files with fixed-length records. For sequential files with variable-length records, the default is 510 bytes. For relative files, the default is 480 bytes.^cEnabled if the file is opened for input, otherwise disabled.

Table 5.3. Summary of IBM Dialect ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
BLKSIZE (expression)	Specifies the size of a block when writing to a buffered file.	Create	Record	Current record size	FIXED BINARY(31)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
CONSECUTIVE	Specifies that records in the file are placed in physical sequence.	Create	Record Stream	Enabled	BIT(1)
F FB FS FBS	Specifies a file with fixed-length records of a maximum record size. These are all equivalent to F; the B and S portions are ignored.	Create	Record	Disabled	BIT(1)
RECSIZE	Specifies the maximum size valid for records in the file.	Create	Record Stream	Varies ^a	FIXED BINARY(31)
SCALARVARYING	Specifies that varying character strings are read/written using the entire storage of the variable.	Create	Record	Disabled	BIT(1)
V VB VS VBS	Specifies a file with variable-length records with a maximum record size. These are all equivalent to V; the B and S portions are ignored.	Create	Record	Enabled	BIT(1)

^a512 bytes for sequential files with fixed-length records. For sequential files with variable-length records, the default is 510 bytes. For relative files, the default is 480 bytes.

Some environment options are accepted but ignored by the Endless PL/I for POSIX Systems compiler, for compatibility with other implementations of PL/I. Table 5.4, “ENVIRONMENT Options That Are Ignored” lists these options:

Table 5.4. ENVIRONMENT Options That Are Ignored

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
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
BLOCK_ BOUNDARY_ FORMAT	Indicates that records must not cross block boundaries.	Create	Record Stream	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(31)
BUCKET_ SIZE (expression)	Defines the number of 512-byte blocks in a bucket for an indexed sequential or a relative file.	Create	Record	Maximum record size	FIXED BINARY(31)
CARRIAGE_ RETURN_ FORMAT	Indicates that records in the file will be printed with default carriage control.	Create	Record	Enabled	BIT(1)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
CONTIGUOUS	Specifies that an output file must be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CONTIGUOUS_ BEST_TRY	Requests that if possible an output file be placed in a physically contiguous extent on disk.	Create	Record Stream	Disabled	BIT(1)
CREATION_ DATE (variable)	Overrides default creation date of the file.	Create	Record Stream	Current date and time	BIT(64) ALIGNED
DEFERRED_ WRITE	Requests file system optimization of output.	Create Open	Record	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(31)
FILE_ SIZE (expression)	Defines the initial number of blocks to be allocated for a file.	Create	Record Stream	Not applicable	FIXED BINARY(31)
INDEX_ NUMBER (expression)	Specifies the initial index to use in accessing records in an indexed sequential file.	Create Open	Record	Disabled	FIXED BINARY(31)
INDEXED	Defines an indexed sequential file.	Open	Record	Disabled	BIT(1)
INITIAL_ FILL	Specifies that the initial fill value specified when the file was created is to be used.	Open	Record	Enabled	BIT(1)
MAXIMUM_ RECORD_ NUMBER (expression)	Specifies the largest record number that will be valid for records in a relative file.	Create	Record	0	FIXED BINARY(31)
MULTIBLOCK_ COUNT (expression)	Specifies the number of blocks to be allocated for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
MULTIBUFFER_ COUNT (expression)	Specifies the number of buffers to be allocated for file system buffering.	Create Open	Record	Current process default	FIXED BINARY(31)
NO_ SHARE	Prohibits all types of shared access to the file.	Open	Record ISAM only	Enabled	BIT(1)

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
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)
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(31)
REVISION_DATE (variable)	Overrides the default revision date of the file.	Close	Record Stream	Date and time file is closed	BIT(64) ALIGNED
SHARED_WRITE	ALLOWS OTHER USERS TO READ AND WRITE RECORDS IN THE FILE.	create open	Record ISAM only	Disabled	BIT(1)
SUPERSEDE	Replaces an existing file with the same file name, file type, and version number.	Create	Record Stream	Enabled	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)
TRUNCATE	Specifies that any unused space allocated for a file be deallocated when the file is closed	Create Open Close	Record	Enabled	Bit(1)
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)

5.3. APPEND Option

VALID DIALECTS: ANSI, DEC

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 or overwrite an existing file.
- 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.

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, you can open the file as follows:

```
DECLARE LISTFILE FILE SEQUENTIAL OUTPUT;
:
OPEN FILE(LISTFILE) ENVIRONMENT(APPEND);
```

This OPEN statement opens the file LISTFILE. The file is positioned at its current end-of-file.

5.4. BLKSIZE Option

VALID DIALECTS: IBM

The BLKSIZE option sets the size of a buffer in BUFFERED file. This option is valid only with BUFFERED files and is ignored otherwise. The format of this option is:

```
BLKSIZE(size)
```

SIZE

A positive integer or a fixed binary(31) static variable specifying the maximum block size for the file. Valid values range from the current record size to 32767 bytes. Note that the actual length of a variable-length record includes the four control bytes containing length information.

Although PL/I provides a default of the current record size, this default will not give you any of the advantages of buffered I/O. To improve performance when using buffers to write to a file, specify a block-size that is a multiple of the record size plus any control bytes.

5.5. CONSECUTIVE Option

VALID DIALECTS: IBM

The CONSECUTIVE option specifies that records in a file are placed in physical sequence. Once the location of one record is determined, the next record location can be determined by its physical position in the file.

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

RULES

- The CONSECUTIVE option is valid when a file is created.

USAGE

The CONSECUTIVE option is enabled by default; you need not specify it explicitly.

5.6. DEFAULT_FILE_NAME Option

VALID DIALECTS: ANSI, DEC

The DEFAULT_FILE_NAME option specifies default file name for a file. Its format is:

```
DEFAULT_FILE_NAME [ (character-expression) ]
```

character-expression

Is a character-string expression.

RULES

- The DEFAULT_FILE_NAME option is meaningful when a file is created or opened.
- The *character-expression* may optionally containing an asterisk (*). If there is no asterisk in the character expression, an asterisk is assumed to be at the rightmost end. The maximum length of the string is 128 characters.
- The value of the TITLE option is inserted in the position of the (assumed) asterisk.

The next example shows how to open the file /usr/users/tim/employee.dat:

```
OPEN FILE (INFILE) INPUT TITLE('employee')
  ENVIRONMENT(DEFAULT_FILE_NAME('/usr/users/tim/*.dat'));
```

The following example shows how to open the file /tmp/workfile:

```
OPEN FILE (INFILE) INPUT TITLE('workfile')
  ENVIRONMENT(DEFAULT_FILE_NAME('/tmp/'));
```

5.7. DELETE Option

VALID DIALECTS: ANSI, DEC

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

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

RULES

- 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.
- The DELETE option can not be used on an indexed sequential file.

USAGE

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 file associated with the PL/I file constant INFILE is deleted.

5.8. F, FB, FS, or FSB Option

VALID DIALECTS: IBM

The F, FB, FS, and FSB options all equate to the dec/ansi dialect FIXED_LENGTH_RECORDS option. Blocking and spanning are unsupported. The format of this option is:

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

See [Section 5.11, “FIXED_LENGTH_RECORDS Option”](#) for more information.

5.9. FILE_ID Option

VALID DIALECTS: ANSI, DEC

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:

```
FILE_ID(variable-reference)
```

variable-reference

Specifies the name of the 6-element array variable containing the file identification. See [Section 5.10, “FILE_ID_TO Option”](#) for information on obtaining the file identification when you create a file.

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

5.10. FILE_ID_TO Option

VALID DIALECTS: ANSI, DEC

Use the FILE_ID_TO option to obtain the file identification when you create a file. Its format is:

```
FILE_ID_TO(variable-reference)
```

variable-reference

Specifies the name of the 6-element array variable. PL/I returns the file identification of the created file to this variable.

You must declare the variable as (6) FIXED BINARY(31), and it must be connected.

RULES

- The FILE_ID_TO option is meaningful when a file is created or opened.
- FILE_ID_TO option conflicts with the FILE_ID option.
- This option can only be used 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.

5.11. FIXED_LENGTH_RECORDS Option

VALID DIALECTS: ANSI, DEC

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:

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

RULES

- The FIXED_LENGTH_RECORDS option must be specified at both creation and open time.
- The FIXED_LENGTH_RECORDS option conflicts 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 provides a default length of 512 bytes for stream files and a default length of 480 bytes for relative files. There is no default for indexed files.

5.12. GROUP_PROTECTION Option

VALID DIALECTS: ANSI, DEC

The GROUP_PROTECTION option defines the type of access to be permitted to the file by other users in the owner's group (also see id(1)). The format of this option is:

```
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:

r—Read access is allowed.

w—Write/delete access is allowed.

x—Execute access is allowed.

Letters can be repeated, but the maximum length of the string is 4 characters. The *character-expression* string is case insensitive; "W" and "w" are equivalent.

For compatibility with OpenVMS implementations of PL/I, you can use "d" as a synonym for "w", and "e" as a synonym for "x". 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 applies the current system or process default protection (see `umask(1)`). If any protection options are specified, the protection for unspecified user categories defaults to no access.

5.13. MAXIMUM_RECORD_SIZE Option

VALID DIALECTS: ANSI, DEC

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.

The format of this option is:

```
MAXIMUM_RECORD_SIZE(integer-expression)
```

integer-expression

Is a numeric expression with values in the range 1 to 32,767. 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 when a file is created and when it is opened. If the specified opened size does not match the specified created size, the results are undefined. If not specified, PL/I provides a default length based on the file organization and record format as follows:

Table 5.5. Default Record Sizes

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

5.14. OWNER_PROTECTION Option

VALID DIALECTS: ANSI, DEC

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 (UID). Refer to `id(2)` for more information. The format of this option is:

```
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. The character-string expression can contain any of the following letters to indicate the access allowed:

r—Read access is allowed.

w—Write/delete access is allowed.

x—Execute access is allowed.

Letters can be repeated, but the maximum length of the string is 4 characters. The *character-expression* string is case insensitive; "W" and "w" are equivalent.

For compatibility with OpenVMS implementations of PL/I, you can use "d" as a synonym for "w", and "e" as a synonym for "x". 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 (see `umask(1)`). If any protection options are specified, the protection for unspecified user categories defaults to no access.

5.15. RECORD_ID_ACCESS Option

VALID DIALECTS: ANSI, DEC

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

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

RULES

- The RECORD_ID_ACCESS option is meaningful when a file is created or opened.

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. See [Chapter 6, Input/Output Statement Options](#) for more information.

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.

5.16. RECSIZE Option

VALID DIALECT: IBM

The RECSIZE 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 a stream file, the RECSIZE option is the same as specifying a LINESIZE on the OPEN statement. Note that specification of LINESIZE on the OPEN statement overrides a prior specification of RECSIZE on the declaration.

The format of this option is:

```
RECSIZE(integer-expression)
```

integer-expression

Is a numeric expression with values in the range 1 to 32,767. 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

If RECSIZE is not specified, PL/I provides a default length based on the file organization and record format as follows:

Table 5.6. Default Record Sizes

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

5.17. SCALARVARYING Option

VALID DIALECTS: ANSI, DEC, IBM

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.

5.18. SHARED_READ Option

VALID DIALECTS: ANSI, DEC

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:

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

RULES

- The SHARED_READ option is meaningful when a file is created or opened.
- The SHARED_READ option 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.

5.19. SUPERSEDE Option

VALID DIALECTS: ANSI, DEC

The SUPERSEDE option specifies that if a file already exists with the same name and type, it will be deleted and a new file created in its place. This is the default. The format of this option is:

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

RULES

- The SUPERSEDE option is meaningful only when a file is created.

- The SUPERSEDE option conflicts with the APPEND option.

COMPATIBILITY NOTE

- On VAX VMS, the default is not to supersede.

USAGE

By default, the file system creates a new file whenever a file is opened for output.

5.20. TEMPORARY Option

VALID DIALECTS: ANSI, DEC

The TEMPORARY option creates a temporary file name using tempnam(3s). The format of this option is:

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

RULES

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

USAGE

When you create a file with the TEMPORARY option, the file system calls tempnam(3s) to create a uniquely named temporary file. A file thus created can be used during the execution of the program and deleted on completion, without cluttering up the current working directory.

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 the name of the temporary file is not available to a PL/I application, 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(31);
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.

5.21. V, VB, VS, or VSB Option

VALID DIALECTS: IBM

The V, VB, VS, and VSB options all equate to the variable length record format, which is the default record format for all dialects. Blocking and spanning are unsupported.

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

RULES

- The V option is valid when a file is created.

USAGE

The variable record format is enabled by default; you need not specify it explicitly.

5.22. WORLD_PROTECTION Option

VALID DIALECTS: ANSI, DEC

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 are not the superuser. Refer to id(1). The format of this option is:

```
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:

r—Read access is allowed.

w—Write/delete access is allowed.

x—Execute access is allowed.

Letters can be repeated, but the maximum length of the string is 4 characters. The *character-expression* string is case insensitive; "W" and "w" are equivalent.

For compatibility with OpenVMS implementations of PL/I, you can use "d" as a synonym for "w", and "e" as a synonym for "x". 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 applies the current system or process default protection (see `umask(1)`). If any protection options are specified, the protection for unspecified user categories defaults to no access.

5.23. ENVIRONMENT Options for File Protection and File Sharing

This section discusses the ENVIRONMENT options that take advantage of special processing options for file protection and file sharing.

5.23.1. File Protection

Each user who is authorized to use the system is assigned a user identification (UID) and a group id (GID) by the system manager. When a PL/I program creates a file, the current UID associated with the process executing the program defines the file's ownership.

Based on this UID, called the owner UID, the file system defines the file protection in terms of which users on the system can access the file and what operations they can perform on the file. These users are defined as follows:

- Owner—Any other process that has the same UID as the file's owner is also the owner of a file.
- Group—A process that has the same GID is a member of the owner's group.
- World—All processes that do not fall into the other categories belong to the world category.

The types of permissions defined for a file are:

- Read access—the right or permission to perform an input operation
- Write access—the right or permission to perform an output, update, or delete operation
- Execute access—the right or permission to execute an executable file

The types of permissions defined for a directory are:

- Read access—the right or permission to read the directory and see information about the files it contains
- Write access—the right or permission to create or delete files in the directory
- Execute access—the right or permission to see information about files you know about

The difference between read access and execute access is that you are allowed to perform wildcard operations on a directory you have read access to, but you must know the exact pathname of a file in a directory to which you have only execute access.

You can explicitly control the protection applied to a file by specifying the type of access allowed each category of user.

In a PL/I program, you can specify file protection when you create the file.

5.23.2. Defining File 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

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 (or d)—gives the right to modify or delete the file.

x (or e)—gives the right to execute executable files.

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 (umask) for all the categories.

For example:

```
ENVIRONMENT (
  OWNER_PROTECTION ('rwx')
  GROUP_PROTECTION('w')
  WORLD_PROTECTION ('r')
```

This specification defines protection to a file as follows:

- The OWNER_PROTECTION option specifies rwx, that is, read, write, delete and execute access. This gives the owner permission to do anything to the file.
- The GROUP_PROTECTION option specifies only write access for group users.
- The WORLD_PROTECTION option specifies only read access.

Note that the UNIX command `chmod` lets the owner of a file change file permissions at any time. Additional commands and user privileges let users override or change file permissions. For details on these commands and privileges, see `umask(1)`.

The file system applies the permissions you specify for a file when the file is accessed from a program or from the command level. It also applies the permissions when the file is shared.

5.24. File Sharing

PL/I lets multiple programs access records in the same file concurrently. All files are opened for `SHARED_READ` and access is controlled by the UNIX-level permissions on the file. Any number of programs can read records in a PL/I file providing they have access to the file.

Input/Output Statement Options

Note

The **OPTIONS** keyword is available in the dec and ansi dialects of Endless PL/I for POSIX Systems only.

PL/I lets you specify the **OPTIONS** keyword on input/output statements and supports certain options for each statement. This chapter explains how to code options for I/O statements, summarizes the valid options for each I/O statement, and describes each option individually.

6.1. Option Format

Use the **OPTIONS** keyword and an options list in a statement to specify I/O statement options. 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 you specify the option in the option list and do not specify a Boolean expression, the default value of 'true' is supplied.

6.2. Summary of Input/Output Statement Options

Table 6.1, “Summary of Input/Output Statement Options” lists the I/O options, their uses, the statements for which they are valid, and their data types.

Table 6.1. Summary of Input/Output Statement Options

Option	Usage	Valid Statements	Data Type
CANCEL_CONTROL_O ^a	Disables effect of CTRL/O prior to terminal output.	PUT	BIT(1)
FAST_DELETE ^b	Deletes a record without updating alternate indexes.	DELETE	BIT(1)

Option	Usage	Valid Statements	Data Type
NO_ECHO	Suppresses display of input data on a terminal.	GET	BIT(1)
NO_FILTER	Sets UNIX raw mode (see stty(1) for more information)	GET	BIT(1)
PROMPT (expression)	Writes a prompting message prior to an input operation.	GET	CHAR(*)
PURGE_TYPE_AHEAD	Clears the terminal type-ahead buffer before reading input data.	GET	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)

^aIgnored; UNIX does not recognize Ctrl/O.

^bIgnored; RMS performance option.

6.2.1. 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

Use the NO_ECHO option when you want to keep data entered at a terminal from being seen by users other than the one who entered the data. For example, if you request a password, 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.

6.2.2. PROMPT Option

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

```
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 will be as:

```
Enter number: 44 Return
```

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

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.

6.2.3. PURGE_TYPE_AHEAD Option

The PURGE_TYPE_AHEAD option specifies, when the input device is a terminal, 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.

6.2.4. 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:

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) be connected.

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 a 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.

6.2.5. RECORD_ID_TO Option

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

RECORD_ID_TO (*variable-reference*)

variable-reference

Is a reference to a 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 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.

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 two built-in file-handling subroutines. They are called built-in subroutines because you do not need to declare them before using them in a PL/I program. The subroutines are:

- FLUSH—Requests the file system to write all buffers onto disk to preserve the current status of a file
- REWIND—Positions a file at its beginning

Both of these subroutines are valid only in the dec dialect.

The following sections describe the FLUSH and REWIND routines.

7.1. FLUSH Built-In Subroutine (DEC Dialect Only)

The FLUSH built-in subroutine writes all file 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:

```
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 PLI to write all internal file buffers back to the file. This subroutine is called implicitly by the REWIND built-in subroutine.

7.2. REWIND Built-In Subroutine (DEC Dialect Only)

The REWIND built-in subroutine positions a file so that the next record to be read will be the first record in the file. 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. The position of the file following the call to the REWIND subroutine is:

- A sequential file is positioned at its first record.
- A relative record file is positioned at the first occupied cell.

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/D 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:

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

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

Using Sort Subroutines (IBM Only)

The `ibm` dialect of Endless PL/I for POSIX Systems provides the IBM sort subroutines, `PLISRTA`, `PLISRTB`, `PLISRTC` and `PLISRTD`. These routines rely on the POSIX-compatible `sort` utility that ships as part of the local system.

To use the sort routines, add calls to your program to one of the Sort program's entry points. The call passes information on the fields to be sorted. You must specify information about the data to be sorted, the type of sort to perform, the format and destination of the output, and so on.

The Sort program called by Endless PL/I for POSIX Systems provides a flexible interface. You define the sorting fields in each record and the order of the sort. You can also specify whether the data originates and/or is output to a file or a user-written routine.

The following sections describe how to use the sort routines.

Note

Do not name any of your routines "SORT" if you are using the sort subroutines. To be safe, do not use any upper or lower case variants of the word "sort" for routine names.

8.1. General Information

Before writing the call to the Sort program, you must determine the following:

- Whether you want to sort by the entire record or by specific fields in the record.
- If you are sorting by specific fields, how many, and where in the record they are located.
- What the data types and sizes of the sort fields are.
- The type of sort you want to do, ascending or descending.

This section contains information on the PL/I sorting interface. Understanding the interface will help you decide which routine to use and what to pass to the routine. The sections following [Section 8.2, “Calling the Sort Subroutines”](#) give details on each portion of the call. They contain information on what specifying sort fields, the available sort types, how to determine storage needs, and so on.

8.1.1. How the Sort Program Interface Works

The Sort program is part of the Endless PL/I for POSIX Systems run-time library. It copies each record to a temporary file, then has the system's `sort` command order the records by their sort fields, and finally reads the records back in sorted order. Records whose sort fields are equal stay in the order in which they were read. The temporary files are created in `/tmp` and removed when the sort ends; see [Section 8.2.4, “Specifying Storage”](#).

The Endless PL/I for POSIX Systems Sort interface provides four entry points to the Sort program. Each entry point specifies a different combination of data source and destination, either from and to a file or from and to a subroutine, or a combination of the two. The entry points are described in [Section 8.1.2, “Determining Which Entry Point to Use”](#).

The Sort program handles the complete sort operation. It communicates with the calling program, and any specified user-written input or output routines, by using return codes.

To use the Sort program, insert a CALL statement in your program. The CALL statement specifies one of the four Sort entry points. It must contain the correct arguments for the specified entry point. Once the Sort program is called, it does the following:

1. Gets records from a file or calls the specified user-written input subroutine to get records, using the selection criteria specified in the call.
2. Sorts the records according to the sorting criteria specified in the call.
3. Places the sorted records in a file or calls the specified user-written output routine to handle the records.
4. Passes a return code, indicating success or failure, back to the calling program.

Note that user-written input or output routines become, for all purposes, part of the Sort program for the duration of the sort. The Sort program calls the routines at the appropriate point in processing. Return codes control the flow of control between the Sort program and the input or output routines, telling the Sort program whether it should call the routine again or continue to the next stage in processing.

8.1.2. Determining Which Entry Point to Use

When sorting in PL/I, the data can originate in a file and be output to another file, or it can originate and/or be output to a user-written PL/I subroutine. The simplest use of the Sort program is to sort a file and direct the results into another file. The input file must be in the current directory, and the output file is created in the current directory. By default, the input file is `sortin` and the output file is `sortout`; see [Section 8.2.7, “Specifying a File Prefix”](#).

Using subroutines lets you manipulate the data before and/or after the sort. For example, you could read a file and sort only certain records, or you could print the resulting sorted data. The format of input and output routines is described more fully in [Section 8.3, “Input and Output Routines”](#).

The Sort program interface provides four entry points. The following table lists the entry points to the Sort program and specifies the source and destination of data in each case:

Table 8.1. Entry Points for the Sort Program

Sort Subroutine Entry Point	Data Source	Data Destination
PLISRTA	File	File
PLISRTB	Subroutine	File
PLISRTC	File	Subroutine
PLISRTD	Subroutine	Subroutine

8.2. Calling the Sort Subroutines

Once you know your sort requirements and have decided on a sort routine, you can write the call to the PLISRTx routine. The following sections give the general syntax of the call and details on each required or optional argument.

8.2.1. General Format of the Sort Interface Call

The format of the call to any of the four sort entry points is:

```
CALL PLISRTx (sort-specification, record-specification, maximum-storage, return-code [,input-
routine] [,output-routine] [,file-prefix,message-level] );
```

sort-specification

A character-string expression of the form:

```
'bSORTbFIELDS=(start1,length1,format1,order1[,startn,lengthn,formatn,ordern] )
[,OPTION1[... ,OPTIONn]]b'
```

This is a required argument.

record-specification

A character-string expression of the form:

```
'bRECORDbTYPE=RECTYPE[,LENGTH=(rec-length[,min-length,modal-length])b'
```

This is a required argument.

maximum-storage

A fixed binary (31) expression giving the amount of storage the Sort program may use. It is accepted for compatibility and has no effect; see [Section 8.2.4, “Specifying Storage”](#).

This is a required argument.

return-code

Fixed binary (31) user-supplied variable in which Sort returns a code indicating success or failure. This is a required argument.

input-routine

Name of the user-written PL/I routine that supplies records to the Sort program. Required for PLISRTB and PLISRTD; invalid otherwise.

output-routine

Name of the user-written PL/I routine that receives sorted records from the Sort program. Required for PLISRTC and PLISRTD; invalid otherwise.

file-prefix

Optional. A character-string expression of up to four characters that replaces the default prefix sort in the names of the input and output files.

message-level

Optional. A character-string expression of two characters, NO, AP or CP. It is accepted for compatibility and has no effect; see [Section 8.2.8, “Specifying Message Level”](#).

The following sections discuss each of these arguments in detail.

8.2.2. Specifying the Sort Fields

The *sort-specification* is the first argument to PLISRTx. It has the following syntax:

```
'bSORTbFIELDS=(start1,length1,format1,order1,...startn,lengthn,formatn,ordern)
[OPTION1[...OPTIONn]]b'
```

B

A single blank space, before SORT and between SORT and FIELDS. Exactly one blank is required in each place, and no other blank spaces are allowed before the field list.

START, LENGTH, FORMAT, ORDER

The sorting information specifies the location of each sort field and how to sort by each field. You can specify several sets of sorting information. Together the fields must not exceed 2,000 bytes, counting each ZD, PD, FI and BI field as 8 bytes.

If you specify more than one sort field, the Sort program sorts on each field in order. In other words, it sorts on the first field, then sorts groups of records that have the same first field according to the second field, and so on.

Sort fields can overlap each other.

The following list gives details on each element of the sorting information.

- start** Specifies the starting position in the record, in bytes. The first byte of the record is byte 1. The field must lie wholly within the record length given in the *record-specification*.
- length** The length, in bytes, of the field. The lengths allowed for each format are given below.
- format** The format of the data being sorted. All data passed between PL/I routines and the Sort program is in the form of character strings. The codes given below tell the Sort program what type of data is actually being sorted.

Code	Data Type and Length
CH	character, any length
ZD	zoned decimal, up to 18 digits
PD	packed decimal, 1 – 9 bytes
FI	binary, signed, 1 – 8 bytes
BI	binary, unsigned, 1 – 8 bytes

Code	Data Type and Length
FL	floating-point, 4 or 8 bytes

PD is a FIXED DECIMAL value. FI is a FIXED BINARY value, and FL a FLOAT BINARY value in IEEE single (4 bytes) or double (8 bytes) precision. FI, BI and FL fields are read in this machine's byte order, which is how PL/I stores them in a record, not in the byte order of IBM mainframes. ZD is the form in which this compiler stores a PICTURE value such as 'S9999': an optional leading sign (+ or -) followed by ASCII digits. It is not IBM's zoned decimal, which carries the sign in the last digit.

order The order in which records will be sorted. One of the following:

- A—Ascending. Example: a,b,c or 1,2,3
- D—Descending. Example: c,b,a or 3,2,1

OPTION1... ,OPTIONN

Additional options to the Sort program. Use a comma to separate any options from the preceding arguments and from each other. Do not put blank spaces between options. Any option not listed here causes the sort to fail.

The following list gives details on each additional option.

FILSZ=y	Gives the number of records in the sort, exactly or, preceded by the letter E, approximately. Accepted for compatibility; it has no effect.
SKIPREC=y	Accepted for compatibility, but it has no effect: no records are skipped. To leave records out of a sort, use PLISRTB or PLISRTD with an input routine that does not pass them on.
EQUALS NOEQUALS	Accepted for compatibility. Records with equal sort fields always stay in the order in which they were read, whichever is given.

8.2.3. Specifying Record Information

The *record-specification* is the second argument to PLISRTx. It has the following syntax:

```
'bRECORDbTYPE=RECTYPE[,LENGTH=(rec-length[,min-length,modal-length)]b'
```

B

One or more blank spaces. These blank spaces are mandatory; no other blank spaces are allowed.

RECTYPE

The type of the records. Only fixed-length records are supported:

F	fixed length
---	--------------

The types V (varying length EBCDIC), D (varying length ASCII) and S (stream) are recognized, but a sort with any of them fails. Records of varying length can still be sorted through an input routine; see [Section 8.3.2, “Writing Input Routines”](#).

LENGTH

The LENGTH argument is required. Every record is exactly *rec-length* bytes long. The maximum length you can specify is 32767 bytes.

<i>rec-length</i>	Length of the records to be sorted, in bytes. Required.
,,	two sort arguments not used in the PL/I interface. Required if you specify any of the following arguments.
<i>min-length</i>	The minimum length of varying-length records. Accepted for compatibility; it has no effect. Optional. Must be specified if <i>modal-length</i> is specified.
<i>modal-length</i>	The most common length of varying-length records. Accepted for compatibility; it has no effect. Optional.

8.2.4. Specifying Storage

You can specify storage either by declaring a fixed binary (31) variable and passing the variable to the Sort routine, or by passing a constant value. For example, either one of the following calls will invoke the Sort program PLISRTA to sort a 40-byte character record using 512,000 bytes of storage:

```
DECLARE RETURN_CODE FIXED BIN(31);
CALL PLISRTA(' SORT FIELDS=(1,40,CH,D) ', ' RECORD TYPE=F,LENGTH=(40) ', 512000, RETURN_CODE);
```

or:

```
DECLARE RETURN_CODE FIXED BIN(31);
DECLARE MAXSTORAGE FIXED BIN(31) INITIAL(512000);
CALL PLISRTA(' SORT FIELDS=(1,40,CH,D) ', ' RECORD TYPE=F,LENGTH=(40) ', MAXSTORAGE, RETURN_CODE);
```

You must specify a value for storage, but it has no effect: the sort command manages its own memory, and uses temporary files when the data does not fit.

The Sort program needs space in /tmp for a copy of every record being sorted, plus a line of about twice the length of the sort fields for each record. The sort command may need further temporary space of its own (in /tmp, unless the environment variable TMPDIR names another directory).

The sort command is run through the shell with the environment variable LC_ALL set to C, so that the records are compared byte by byte and not according to the locale. It must be in the search path (PATH).

Only one sort can run at a time in a program. Do not call PLISRTA, PLISRTB, PLISRTC or PLISRTD from an input or output routine.

8.2.5. Using the Return Code

The required return code argument is a variable to contain the fixed binary(31) value returned by the sort program. The sort program returns one of the following:

- 0—success
- 16—failure

In your calling program, declare a fixed binary(31) variable to contain return code.

To determine success or failure of the sort, check the return code after the call to the Sort program. If the *sort-specification* or *record-specification* is not valid, the Sort program also writes a message ending in “in SORT statement **** Execution aborted” to standard output and raises the ERROR condition; if an ON-unit for ERROR returns, the return code is 16.

For example, assume that the variable RETURN_CODE was declared for use with the Sort program. The following code calls PLISRTD to sort 80-byte records on a single 10-character field at the start of the record. The input and output routines are SORT_IN and SORT_OUT. After returning from the call, the code checks the success of the sort:

```
CALL PLISRTD (' SORT FIELDS=(1,10,CH,A) ',
             ' RECORD TYPE=F,LENGTH=(80) ',
             256000,RETURN_CODE,SORT_IN,SORT_OUT);
IF RETURN_CODE ^= 0 THEN
DO;
    PUT SKIP LIST ('Error! Return code = '||RETURN_CODE);
    SIGNAL ERROR;
END;
```

8.2.6. Specifying Input and Output Routines

The *input-routine* argument is used with PLISRTB and PLISRTD. The *output-routine* argument is used with PLISRTC and PLISRTD. These arguments are required for the specified routines. They give the name of the user-written PL/I routine that supplies records to or accepts records from the Sort program. See [Section 8.3, “Input and Output Routines”](#) for more information on writing routines for use with the Sort program. See the preceding section for an example of calling PLISRTD, specifying the user-written input routine SORT_IN and the user-written output routine SORT_OUT.

8.2.7. Specifying a File Prefix

The optional file prefix argument lets you replace the default prefix sort in the names of the input and output files the Sort program uses. Using a file prefix lets you sort several files in turn without renaming them. For example, you can specify 'TEMP' for the file prefix, as shown in the following example:

```
CALL PLISRTA(' SORT FIELDS=(1,40,CH,D) ', ' RECORD TYPE=F,LENGTH=(40)
',STOR,RETURN_CODE,'TEMP');
```

In this case, PLISRTA reads TEMPIn and writes TEMPout. The prefix is used exactly as given, followed by in or out in lower case; by default the files are sortin and sortout. The temporary files in /tmp are named after the prefix too.

The file prefix is at most four characters; a blank ends it early. A blank or null string selects the default prefix, sort.

If you omit this argument but require the *message-level* argument, specify a null string ('') in place of the file prefix argument.

8.2.8. Specifying Message Level

The *message-level* argument is accepted for compatibility with other implementations of PL/I, where it controls the Sort program's diagnostic messages. Here the Sort program writes no such messages, so it has no effect. It must still be one of the following, or the sort fails; AP is assumed if it is omitted:

- NO—no messages.
- AP—all messages.
- CP—critical messages only.

If you omit the *file-prefix* argument but require the *message-level* argument, specify a null string ('') in place of the file prefix argument. For example, the following call invokes PLISRTB to sort on the first 12 characters of a 30 character record in ascending order. Input is passed to the Sort program by a user-written routine called READ_DATA, the file prefix is defaulted to sort, and no messages are passed back:

```
CALL PLISRTB(' SORT FIELDS=(1,12,CH,A) ', ' RECORD TYPE=F,LENGTH=(30)
',MAXSTORAGE,RETURN_CODE, READ_DATA,' ','NO');
```

8.3. Input and Output Routines

When sorting in PL/I, the data can originate in a file and be output to another file. This is done using PLISRTA, the simplest interface.

Input and output routines give you additional flexibility. When using routines PLISRTB, PLISRTC, and PLISRTD, your sort data can originate and/or be output to a user-written routine. These input and output routines are written in PL/I. They can be either internal or external procedures. If they are internal to the calling program (the program routine with the call to the Sort program), the input and output routines follow the usual scoping rules. Whether or not the routines are internal to the calling program, the input and output routine names must be known within the scope of the block containing the call.

The Sort program calls the input and output routines once for each record being sorted. Therefore, write each routine to handle one record at a time. Note that this means automatic variables will not retain their value between calls. If a variable, such as a counter, must retain its value between calls, either declare it STATIC or declare it in the containing block.

8.3.1. Using the PLIRETC Built-In Subroutine

The PLIRETC built-in subroutine sets the PL/I return code. Use PLIRETC in your input and output routines to return a value to the Sort program. The PLIRETV built-in function returns the current value of the PL/I return

code, so a program can also use the two for its own purposes, to pass a value from one procedure to another. Both are available in the ibm dialect only.

The format of the subroutine call is:

```
CALL PLIRETC(return-code)
```

return-code

An expression, converted to fixed binary(15). It becomes the PL/I return code.

The PL/I return code is zero when the program starts, and keeps the value of the most recent call to PLIRETC, in any procedure. The Sort program sets it as well: to 4 before it passes the first record to an output routine, so an output routine that does not call PLIRETC goes on receiving records. The PL/I return code is not passed to the operating system when the program ends; it is not the program's exit status.

For example, the following procedure records whether it found any errors, and its caller tests the result:

```
CHECK_INPUT: PROCEDURE;
...
IF ERRORS_FOUND THEN CALL PLIRETC(8);
                  ELSE CALL PLIRETC(0);
END CHECK_INPUT;

CALL CHECK_INPUT;
IF PLIRETV ^= 0 THEN PUT SKIP LIST ('Errors in input.');
```

8.3.2. Writing Input Routines

Use an input routine to manipulate the data in some way before sorting it. For example, you might want to print the data or generate or modify the sorting fields.

The input routine and the Sort program communicate using calls to PLIRETC. The Sort program recognizes two PLIRETC return codes from an input routine:

- 8—All records have been sent. The Sort program ignores any record returned by your input routine when the return code is 8.
- 12—Include this record in the sort.

Any other return code is treated as 12: the record is included.

Calls made to PLISRTB or PLISRTD require an input routine. When using an input handling routine, the sort processing is as follows:

1. Your program calls PLISRTB or PLISRTD.
2. The Sort program assumes control and calls the input routine.
3. The input routine assumes control and does the following:
 - a. The input routine checks to see if there are any more records to sort. If not, it calls PLIRETC with a *return-code* of 8.

- b. If there are records, the input routine does any processing required, then calls PLIRETC with a *return-code* of 12, and RETURNS, passing back a record in character-string format.
4. The Sort program assumes control again and processes the record. It continues to call the input routine and receive records until it receives a return code of 8.
5. The Sort program sorts all the records.
6. The Sort program either writes the sorted records to the output file or calls the output routine, described in the next section.
7. The Sort program sets up the *return-code* and returns to the statement after the CALL PLISRTx statement.

The input routine passes records back to the Sort program in the RETURN statement. Therefore you must specify the RETURNS attribute in the input routine PROCEDURE statement.

Returned records must be in character-string format. They can be fixed or varying. A varying string is padded with blanks to the record length given in the LENGTH argument, so sort-field positions count from the first character of the string. A varying string longer than the record length makes the sort fail.

EXAMPLE

```

SORT_INPUT_ROUTINE: PROCEDURE RETURNS(CHAR(80));
    /* you must specify the returns attribute. For varying
       strings, the length must be the maximum length.

       DECLARE CUST_RECORD CHARACTER(80);
       DECLARE CUSTOMER_FILE FILE RECORD INPUT;

       ON ENDFILE(CUSTOMER_FILE) BEGIN;
           PUT SKIP(2) LIST('End of customer file input. ');
           CALL PLIRETC(8); /* signal end of input to Sort */
           GOTO END_ROUTINE;
       END;

    /* Do any processing you require to get a record. */
    /* For example, you could get the record from a file */
    /* and then display the record. */
    READ FILE (CUSTOMER_FILE) INTO (CUST_RECORD);
    PUT SKIP LIST(CUST_RECORD);
    /* signal Sort to include this record */
    CALL PLIRETC(12);

    RETURN(CUST_RECORD); /* return the record to Sort */

END_ROUTINE:
END SORT_INPUT_ROUTINE;

```

8.3.3. Writing Output Routines

Use an output routine to do any post-sorting processing. For example, you might want to print the data or use the sorted data in a calculation.

The output routine and the Sort program communicate using calls to PLIRETC. The Sort program recognizes one PLIRETC return code from an output routine:

- 4—Pass the next record.

Any other return code stops the Sort program passing records to the output routine; the remaining records are not passed.

Calls made to PLISRTC or PLISRTD require an output routine. When using an output handling routine, the sort processing is as follows:

1. Your program calls PLISRTC or PLISRTD.
2. The Sort program assumes control. When it requires a record, the Sort program gets it from the input file or the input routine, as described in the preceding section.
3. The Sort program processes all of the records.
4. The Sort program calls the output routine, passing it one record.
5. The output routine assumes control and does the following:
 - a. Processes the record as required.
 - b. Calls PLIRETC with a *return-code* of 4.
6. The Sort program assumes control and continues to pass records to the output routine until all the records have been processed.
7. The Sort program sets up the *return-code* and returns to the statement after the CALL PLISRTx statement.

Note that the Sort program does not pass an end-of-data code to your output routine. Do any end-of-data processing after the Sort program returns to the calling program. A record of a CHARACTER VARYING output routine parameter is always passed at the full record length.

EXAMPLE

```
SORT_OUTPUT_ROUTINE: PROCEDURE(CUST_RECORD);

    DECLARE CUST_RECORD CHARACTER(80);
    DECLARE NEWCUST_FILE FILE RECORD OUTPUT;

    /* Do any processing you require. */
    /* For example, you could display the record. */
    PUT SKIP LIST(CUST_RECORD);
    CALL PLIRETC(4); /* request the next record */

    RETURN;
END SORT_OUTPUT_ROUTINE;
```

8.4. Programming Examples

This section contains an example of calling PLISRTA, which uses a file for both input and output, and PLISRTD, which uses input and output routines. PLISRTB and PLISRTC use a combination of files and routines. If you want to use either of those entry points, look at both of the following programs for examples.

8.4.1. Calling the PLISRTA Entry Point

```

SORT_EXAMPLE1: PROCEDURE OPTIONS(MAIN);

/* declare a variable to hold the code returned by the Sort
/* program
*/
DECLARE RETURN_CODE FIXED BIN(31);

/* call the PLISRTA entry point of the sort routine. */

CALL PLISRTA (' SORT FIELDS=(5,20,CH,A,25,20,CH,A,45,1,CH,A) ',
              ' RECORD TYPE=F,LENGTH=(120) ',
              4000000,
              RETURN_CODE,
              'CUST');

SELECT (RETURN_CODE);
  WHEN(0) PUT SKIP LIST ('Sort completed successfully. ');
  WHEN(16) PUT SKIP LIST ('Sort failed. ');
  OTHERWISE
    PUT SKIP LIST('Invalid Sort return code'||RETURN_CODE);
END; /*select*/

END SORT_EXAMPLE1;

```

The example above sorts a customer file by name. The arguments of the call to PLISRTA are:

- SORT FIELDS—There are three sort fields, which are in this case a last name, first name, and middle initial. The fields are all character strings. The sort order is ascending.
- RECORD TYPE—Records are fixed length and 120 bytes long.
- STORAGE—4 Mb, accepted for compatibility; it has no effect.
- RETURN CODE—The variable RETURN_CODE will contain the return code from the Sort program.
- FILE PREFIX—The Sort program reads the file CUSTin and writes CUSTout, instead of using the default sortin and sortout.
- MESSAGE LEVEL—This call does not specify a message level, so AP is assumed.

8.4.2. Calling the PLISRTD Entry Point

```

SORT_EXAMPLE2: PROCEDURE OPTIONS(MAIN);

/* declare a variable to hold the code returned by the Sort
/* program
*/
DECLARE RETURN_CODE FIXED BIN(31);

/* call the PLISRTD entry point of the sort routine. */

CALL PLISRTD (' SORT FIELDS=(5,20,CH,A,25,20,CH,A,45,1,CH,A) ',

```

```

        ' RECORD TYPE=F,LENGTH=(120) ',
        4000000,
        RETURN_CODE,
        SORT_INPUT,
        SORT_OUTPUT,
        'CUST');

SELECT (RETURN_CODE);
  WHEN(0) PUT SKIP LIST ('Sort completed successfully. ');
  WHEN(16) PUT SKIP LIST ('Sort failed. ');
  OTHERWISE
    PUT SKIP LIST('Invalid Sort return code'||RETURN_CODE);
END; /*select*/

SORT_INPUT: PROCEDURE RETURNS(CHAR(120) VARYING);
/* you must specify the returns attribute. For varying
   strings, the length must be the maximum length. */

  DECLARE CUST_RECORD CHARACTER(120) VARYING;
  DECLARE CUSTOMER_FILE FILE RECORD INPUT;

  ON ENDFILE(CUSTOMER_FILE) BEGIN;
    PUT SKIP(2) LIST('End of customer file input. ');
    CALL PLIRETC(8); /* signal end of input to Sort */
    GOTO END_ROUTINE;
  END;

/* Do any processing you require to get a record. */
/* For example, you could get the record from a file */
/* and then display the record. */
  READ FILE (CUSTOMER_FILE) INTO (CUST_RECORD);
  PUT SKIP LIST(CUST_RECORD);
/* signal Sort to include this record */
  CALL PLIRETC(12);

  RETURN(CUST_RECORD); /* return the record to Sort */

END_ROUTINE:
END SORT_INPUT;

SORT_OUTPUT: PROCEDURE(CUST_RECORD);

  DECLARE CUST_RECORD CHARACTER(*);
  DECLARE NEWCUST_FILE FILE RECORD OUTPUT;

/* Do any processing you require. */
/* For example, you could display the record. */
  PUT SKIP LIST(CUST_RECORD);
  CALL PLIRETC(4); /* request the next record */

  RETURN;

```

```
END SORT_OUTPUT;  
  
END SORT_EXAMPLE2;
```

The example sorts the same customer records by customer name, but the records pass through an input and an output routine. The elements of the call to PLISRTD are similar to the ones in the call to PLISRTA. The differences are:

- Varying length records.

The input routine returns CHARACTER(120) VARYING strings. The Sort program pads each one with blanks to the record length, 120, so the sort fields are at the same positions as in the previous example, and the output routine receives each record as a 120-character string.

- The input and output routine names.

The two routines are internal to the main program.

Error Handling

In PL/I, 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 VAXCONDITION condition
- The actions that ON-units can take
- The relationship of UNIX condition handling to PL/I 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 (ONCODE, and ONFILE)

Refer to the Endless PL/I for POSIX Systems Reference Manual for descriptions of the ON, REVERT, and SIGNAL statements and ON conditions.

9.1. 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:

```
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 UNIX operating system environment.

9.2. VAXCONDITION Condition

VAXCONDITION provides a general-purpose signaling mechanism for PL/I applications that execute under the control of the VMS operating system. This condition is carried to the PL/I compiler for compatibility reasons.

An ON-unit for a VAXCONDITION condition is established in the following format:

```
ON VAXCONDITION (expression) on-unit
```

expression

Any expression yielding a FIXED BINARY(31) value that can represent either a specific VMS condition value or status value, or an arbitrary user-specified value. The low-order three bits of the value are interpreted as the condition's severity by the default PL/I and system condition handlers. The expression is evaluated when the ON statement is executed, not when the condition is signaled.

A VAXCONDITION ON-unit can be used in conjunction with calls to system service, run-time library, or other system procedures that return status values. An error-handling routine can test the return status value from a system procedure and, if not successful, signal the error.

ON-units can exist for any number of specific values. For example:

```
VCND: PROCEDURE OPTIONS(MAIN);

%REPLACE COND1 BY 1234;
%REPLACE COND2 BY 5678;
ON VAXCONDITION(COND1), VAXCONDITION(COND2)
BEGIN;
    PUT SKIP LIST ('Main handler for VAXCONDITIONs: ',
                  COND1,COND2);
    PUT SKIP EDIT (' Caught ', oncode()) (A,F(5,0));
END;

P1: PROCEDURE;

ON VAXCONDITION(COND1)
BEGIN;
    PUT SKIP EDIT ('p1: caught VAXCONDITION: ',COND1)
                  (A,F(5,0));
    CALL RESIGNAL ();
END;

SIGNAL VAXCONDITION(COND1);

END P1;

P2: PROCEDURE;

ON VAXCONDITION(COND2)
BEGIN;
    PUT SKIP EDIT ('p2: caught VAXCONDITION: ',COND2)
                  (A,F(5,0));
    CALL RESIGNAL ();
END;

SIGNAL VAXCONDITION(COND2);
```

```
END P2;  
  
    CALL P1;  
    CALL P2;  
    CALL P1;  
  
END VCND;
```

9.3. 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.

9.3.1. 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. Normal return from a CONVERSION condition that modifies ONCHAR or ONSOURCE pseudovariables causes the conversion to be retried.

9.3.2. Resignaling the Condition

In PL/I, 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. The next example and [Figure 9.1, “Resignaling a Condition”](#) illustrate the effect of the RESIGNAL built-in subroutine. The reverse numbers 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.

```

A: PROCEDURE OPTIONS(MAIN); ❶
    ON FIXEDOVERFLOW BEGIN;

    :
    END;

    CALL B;
B: PROCEDURE (X,Y); ❷
    ON ANYCONDITION BEGIN;
        IF (ONCODE() = SIGNAL_FOUND)!
            (ONCODE() = SIGNAL_DONE) THEN
            BEGIN;

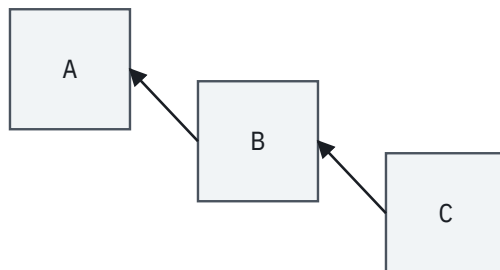
            END; /*Begin block for IF statement */
            ELSE CALL RESIGNAL();
        END; /* ON-unit */
    CALL C;
C: PROCEDURE; ❸

    : /* FIXEDOVERFLOW signaled ❹
    RETURN;
    END;

```

Figure 9.1, “Resignaling a Condition” illustrates the preceding example.

Figure 9.1. Resignaling a Condition



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 9.5, “Search Path for ON-Units”](#).

9.3.3. Executing a Nonlocal GOTO

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.

This removal of call frames from the call stack is called an *unwind*. The next example and [Figure 9.2](#), “Unwinding the Call Stack” illustrates a situation in which an unwind occurs. The reverse numbers indicate the order of execution. The ERROR ON-unit established in procedure A receives control when the ERROR condition is signaled in procedure C. This ON-unit executes the GOTO PRINT_MSG statement. The label PRINT_MSG is in procedure A. Thus, the call stack is unwound and the call frames for the ON-unit, procedure C, and procedure B, in that order, are removed from the stack, and execution continues at the label PRINT_MSG.

```

A: PROCEDURE OPTIONS(MAIN); ❶
    ON ERROR GOTO PRINT_MSG;

    : ❷
    CALL B;
    PRINT_MSG;

    :
B: PROCEDURE; ❸

    CALL C;

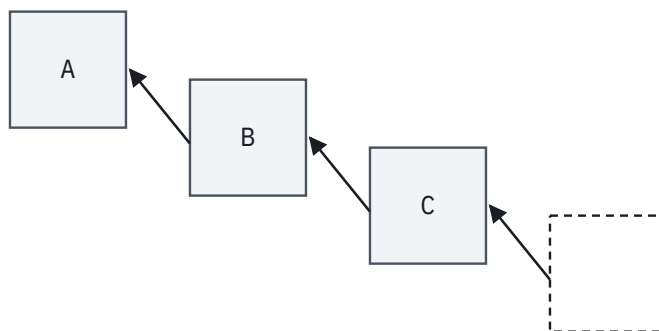
    RETURN;
    END;
C: PROCEDURE; ❹

    RETURN; /* ERROR signaled */
END;

```

Figure 9.2, “Unwinding the Call Stack” illustrates the preceding example.

Figure 9.2. Unwinding the Call Stack



When an unwind occurs in the UNIX environment, PL/I examines each call frame in the calling sequence to determine if a condition ON-unit exists for that frame. If so, PL/I calls the ON-unit with the condition value SS \$_UNWIND, and the ON-unit has the chance to perform block- or procedure-specific cleanup operations.

9.3.4. Stopping the Program

You can use an ON-unit to terminate program execution. To do this, include a STOP statement in the ON-unit. 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 exit(3) system library routine 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 terminates.

Important

If a FINISH or ANYCONDITION ON-unit executes a STOP statement, the program enters an infinite loop. Therefore, do not put STOP statements in FINISH or ANYCONDITION ON-units.

9.4. Relationship of UNIX Signals to PL/I ON-Units

In the UNIX environment, an exception is a hardware- or software-detected condition that synchronously or asynchronously 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 do not provide language constructs to handle exceptions; therefore, condition handlers are established by calling the UNIX Run-Time Library procedure signal(3). The PL/I language, however, uses the ON-unit condition handler to handle a specific condition. By using the keyword condition names defined by PL/I and the extensions provided by PL/I, you can write ON-units to handle any possible UNIX-specific condition. Each procedure can establish separate ON-units for each of several possible conditions that the procedure wishes to handle.

Important

You should never use signal(3) to establish a condition handler in a PL/I call frame.

UNIX signals are mapped to PL/I signals and can be handled by on units in PL/I. The main program establishes default handlers for UNIX signals, thus this behavior is only in affect when the main program is in PL/I. PL/I

runtime library establishes the default signal handler for all UNIX signals except SIGSTOP and SIGKILL which cannot be caught. For more information about UNIX signals see the `signal(3)` manpage.

The default handling of signals is as follows:

1. This signal is translated into an appropriate ONCODE. Refer to [Table 9.1, “Corresponding Values”](#) for mapping from signal to oncode values.
2. The normal condition handling mechanism is used to identify the appropriate PL/I condition handler to be invoked. That is, the stack is searched for an on unit that handles the condition. Floating point exceptions that are generated by hardware conditions are translated one of the following conditions:

OVERFLOW

UNDERFLOW

ZERODIVIDE

All other signals generated are handled either by a `VAXCONDITION(signal)` handler for the specific signal code or by the PL/I `ANYCONDITION` handler. As usual the specific code that caused the condition is available by calling `ONCODE()`.

3. If there is no ON-unit condition handler for the signal, the default action is the default action as defined by the operating system. This is described in `signal(3)`.

Note that unlike other conditions, in the case of a UNIX signal not handled by a specific on condition handler, a PL/I `ERROR` condition is NOT generated. This allows UNIX signals to behave in the normal way if no condition handlers are used.

9.4.1. Accessing the Signal Values

Within a PL/I program the values for the UNIX signals are available in the following variables, each corresponding to one of the UNIX signals. For access to these variables, a program can `GLOBALREF` the variables. The standard include file, `signaldef.in`, contains the declarations for these variables. To include the `signaldef.in` file in your PL/I program, use the following statement:

```
%include $signaldef;
```

[Table 9.1, “Corresponding Values”](#) identifies the correspondence between UNIX signal numbers and the corresponding ONCODE value.

Table 9.1. Corresponding Values

Signal	Variable	Initial Value	
		Decimal	Hexadecimal
SIGHUP	sig\$_hup	140115980	0x85a000c
SIGINT	sig\$_int	1617	0x651
SIGQUIT	sig\$_quit	140115996	0x85a001c
SIGILL	sig\$_ill	140116004	0x85a0024

Signal	Variable	Initial Value	
		Decimal	Hexadecimal
SIGTRAP	sig\$_trap	140116012	0x85a002c
SIGIOT	sig\$_iot	140116020	0x85a0034
SIGEMT	sig\$_emt	140116028	0x85a003c
SIGFPE	sig\$_fpe	140116036	0x85a0044
SIGBUS	sig\$_bus	140116052	0x85a0054
SIGSEGV	sig\$_segv	12	0xc
SIGSYS	sig\$_sys	140116068	0x85a0064
SIGPIPE	sig\$_pipe	140116076	0x85a006c
SIGALRM	sig\$_alarm	140116084	0x85a0074
SIGTERM	sig\$_term	140116092	0x85a007c
SIGURG	sig\$_urg	140116098	0x85a0082
SIGTSTP	sig\$_tstp	140116114	0x85a0092
SIGCONT	sig\$_cont	140116122	0x85a009a
SIGCHLD	sig\$_chld	140116130	0x85a00a2
SIGTTIN	sig\$_ttin	140116138	0x85a00aa
SIGTTOU	sig\$_ttou	140116146	0x85a00b2
SIGIO	sig\$_io	140116154	0x85a00ba
SIGXCPU	sig\$_xcpu	140116164	0x85a00c4
SIGXFSZ	sig\$_xfsz	140116172	0x85a00cc
SIGVTALRM	sig\$_vtalrm	140116180	0x85a00d4
SIGPROF	sig\$_prof	140116188	0x85a00dc
SIGWINCH	sig\$_winch	140116194	0x85a00e2
SIGUSR1	sig\$_usr1	140116212	0x85a00f4
SIGUSR2	sig\$_usr2	140116220	0x85a00fc
SIGTRAP	ss\$_intdiv	1156	0x484
SIGTRAP	ss\$_intovf	1148	0x47c
SIGTRAP	ss\$_subrng	1196	0x4ac
SIGFPE	ss\$_fltovf	1164	0x48c
SIGFPE	ss\$_decovf	1188	0x4a4
SIGFPE	ss\$_fltdiv	1172	0x494

Signal	Variable	Initial Value	
		Decimal	Hexadecimal
SIGFPE	ss\$_fltund	1180	0x49c

9.4.2. Modifying the ONCODE Values

The numbers listed in Table 9.1, “Corresponding Values” for the ONCODE values are under program control. The values listed are the initial value for each of the signals. These numbers may be modified, if desired, by assigning a new value to the variable. Take care not to duplicate these signal numbers or they cannot be distinguished. These values should not be modified within a condition handler.

You may want to use your own number for the ONCODE value, if the numbers are used to identify a related error message.

The following program shows an example of how UNIX signals can be handled, and how the ONCODE values can be modified:

```
UNIXSIG: PROC OPTIONS( MAIN );

dcl sig$_usr1 fixed bin(31) globalref;
dcl sig$_int fixed bin(31) globalref;
dcl sig$_quit fixed bin(31) globalref;
/* set specific value for sig$_int */
sig$_int = 999999;

call mainloop();

/* set up anycondition handler to test for
 * sig$_usr1, sig$_int and sig$_quit */
mainloop: proc;
on anycondition begin;
    if (oncode() = sig$_usr1) then
        PUT LIST ( 'mainloop: caught sigusr1 ',oncode());
    else if oncode() = sig$_int then
        PUT LIST ( 'mainloop: caught sigint ',oncode());
    else if oncode() = sig$_quit then
        PUT LIST ( 'mainloop: caught sigquit ',oncode());
    else call resignal();
    put skip;
end;

/* set up specific handler for sig$_usr2 */
on vaxcondition(sig$_usr2) begin;
    PUT LIST ( 'usr2: caught sigusr2 ',oncode());
    put skip;
end;

/* main loop processing starts here */
:
END;
```

In this example, specific handlers are set up for the UNIX signals SIGUSR1, SIGINT, SIGQUIT, and SIGUSR2. The first three are caught in the ANYCONDITION handler and the ONCODE is compared to the current values in the global variables sig\$_usr1, sig\$_int and sig\$_quit. The signal SIGUSR2 is caught in a specific VAXCONDITION on units testing for sig\$_usr2.

9.5. Search Path for ON-Units

In PL/I, each call frame in the calling sequence for ON-units is searched 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.

The next example and [Figure 9.3, “Execution of an ON-Unit”](#) illustrates the call frames established for the execution of a series of procedures. 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.

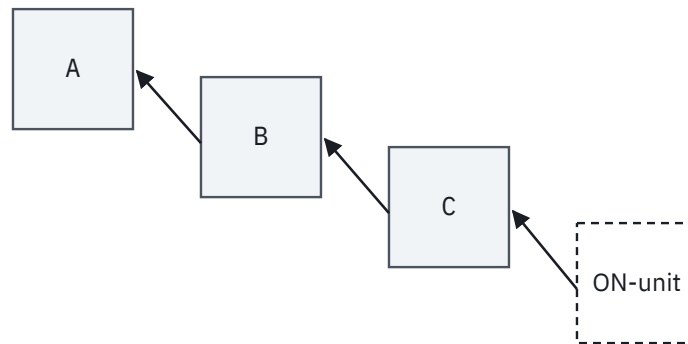
```
A: PROCEDURE OPTIONS(MAIN);
    ON FIXEDOVERFLOW BEGIN;
    :
    END;
    CALL B;

B: PROCEDURE;
    ON UNDEFINEDFILE (PRINTFILE) OPEN
        FILE (PRINTFILE) TITLE('SYS$OUTPUT');
    :
    CALL C;

C: PROCEDURE;
    :
    RETURN; /* Fixed overflow signaled */
    END;
```

Figure 9.3, “Execution of an ON-Unit” illustrates the preceding example.

Figure 9.3. Execution of an ON-Unit



9.5.1. Default Handling for Main Procedures

If PL/I entered a program at a procedure containing 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 ENDPAGE condition is for SYSPRINT, no action is performed.
- 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 UNIX condition handler. This handler prints a message and terminates the program.
- 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 UNIX condition handler prints a message. If the error is a fatal error, the default handler terminates the program; if the error is nonfatal, the program continues.

9.5.2. 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:

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 which repeats steps one and two. If the resignal results in the return of control to the system, the default UNIX condition handler issues a message. If the error was a fatal error, the default UNIX handler terminates the program. Otherwise, the program continues.

The next example and [Figure 9.4, “Search for an ON-Unit”](#) illustrate a search for an ON-unit that will handle a decimal overflow condition. The reverse numbers indicate the sequence in which the search takes place.

```

A: PROCEDURE OPTIONS(MAIN);
    ON ERROR BEGIN; ❹ /* ERROR signaled */

        : ❸ /* Execute ON-unit */
    END;
    CALL B; ❺
    :
B: PROCEDURE; ❻ /* Look for ERROR */
    :
    CALL C;
    :
    RETURN; ❷
    END;

C: PROCEDURE; ❼ /* Look for ERROR */
    :
    RETURN; ❶
    END;

```

Referring to the example above and [Figure 9.4, “Search for an ON-Unit”](#), you can see that PL/I takes the following actions:

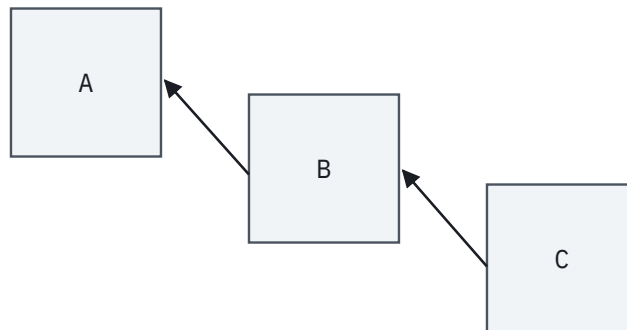
- ❶ Searches the call frame (C) in which the condition was signaled for three specific ON-units: VAXCONDITION (SS\$_DECOVF), FIXEDOVERFLOW, ANYCONDITION

- ② Searches the previous call frame (B)
- ③ Reaches the main procedure (A) ON-units
- ④ Signals the ERROR condition
- ⑤ Searches the call frame (C) ERROR ON-unit

Searches the previous call frame (B)

- ⑦ Locates the ERROR ON-unit in the main procedure (A)

Figure 9.4. Search for an ON-Unit



9.5.3. 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.

The next example and [Figure 9.5, “Effect of Multiple Conditions”](#) illustrate the search sequence followed when a second condition occurs during the execution of an ON-unit. The reverse numbers indicate the order of execution. The ERROR condition in procedure C is handled by the ON-unit established in procedure B. During the execution of this ON-unit, a FIXEDOVERFLOW condition is signaled. PL/I locates the ON-unit (shown in the [Figure 9.5, “Effect of Multiple Conditions”](#) as a box with dotted lines) established for FIXEDOVERFLOW conditions in procedure C and gives it control.

```

A: PROCEDURE OPTIONS(MAIN); ①
    :
    CALL B;

B: PROCEDURE; ②
    ON ERROR BEGIN;

    :
    END;
    CALL C;

C: PROCEDURE; ③ /* Original ERROR signaled */ ④
    ON FIXEDOVERFLOW BEGIN;
  
```

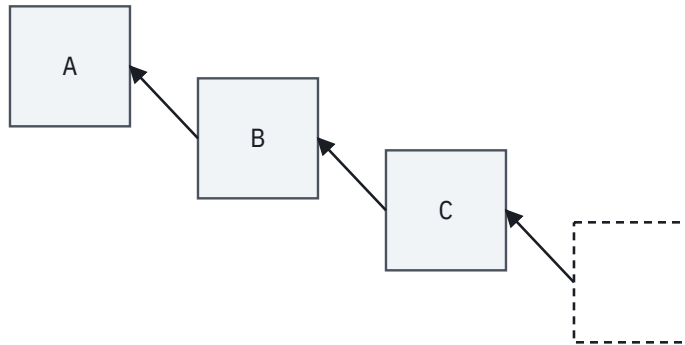
```

:
END;

```

Figure 9.5, “Effect of Multiple Conditions” illustrates the preceding example.

Figure 9.5. 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; the program will enter an infinite loop when the STOP statement executes. The STOP statement signals FINISH, the current ON-unit is re-executed, 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('0'B));
END;

```

Although it is not specifically resigaled, 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 re-executed.

9.6. 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.

9.7. 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.

The following example shows 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.

The following example shows 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.

See [Chapter 4, File Organizations and Input/Output](#) demonstrates an ON-unit that handles errors encountered during record I/O operations.

9.8. Condition-Handling Built-In Functions

The following sections discuss the two PL/I built-in functions that are useful for condition handling: ONCODE, and ONFILE.

9.8.1. ONCODE Built-In Function

All UNIX-defined conditions have symbolic names associated with them. When the PL/I ONCODE built-in function is invoked in an ON-unit, it returns the value of the associated global symbol. This 32-bit status value describes the condition that was signaled. The low-order three bits of the status value contain the severity of

the condition (success, warning, error, or fatal). Severity is important when no ON-unit exists for a condition, and default condition handling is performed by PL/I (see [Section 9.5, “Search Path for ON-Units”](#)). This number is program dependent.

All symbolic names associated with UNIX 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 attribute, 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. You can use the system global symbol names to refer to the values returned by the ONCODE built-in function in an condition-handling ON-unit.

[Table 9.2, “ONCODE Values for PL/I ON Conditions”](#) lists the PL/I keyword condition names and the UNIX global symbol names associated with each condition name.

Table 9.2. ONCODE Values for PL/I ON Conditions

PL/I Condition	UNIX Global Symbol Name
AREA ^b	See the Endless PL/I for POSIX Systems Reference 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 ^c
FINISH	PLI\$_FINISH
FIXEDOVERFLOW	SS\$_DECOVF or SS\$_INTOVF
KEY	PLI\$_KEY
OVERFLOW	SS\$_FLTQVF
STORAGE	PLI\$_STORAGE
STRINGRANGE ^b	PLI\$_STRANGE
SUBSCRIPTRANGE ???	PLI\$_SUBRANGE
UNDEFINEDFILE	PLI\$_UNDEFINEDFILE
UNDERFLOW	SS\$_FLTUND
VAXCONDITION	Any user-defined condition or UNIX signal global symbol value that was signaled
ZERODIVIDE	SS\$_FLTDIV or SS\$_INTDIV

^aIf 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.

^bOnly signalled via SIGNAL statement. No hardware support on UNIX.

^cThese names correspond to the identification fields in the run-time messages.

9.8.2. 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 Endless PL/I for POSIX Systems Reference Manual.

The ONFILE built-in function can be used in 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.

9.8.3. 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 Endless PL/I for POSIX 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.

Using PL/I in the Common Language Environment

The PL/I compiler is part of the UNIX Standards 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. You can call any of the following routine types from PL/I:

- Routines written in other UNIX languages that conform to the common language environment
- UNIX Run-Time Library routines
- UNIX system services
- UNIX utility routines

The terms routine, procedure, and function are used throughout this chapter.

Important

The definitions of routine, procedure, and function used in this chapter differ somewhat from standard PL/I terminology. The following definitions, used for consistency with the UNIX system routines documentation, apply only to this chapter:

- routine—a closed, ordered set of instructions that performs one or more specific tasks. Every routine has an entry point (the routine name), and an optional argument list. Procedures and functions are specific types of routines.
- procedure—a routine that does not return a value.
- function—routine that returns a value.
- system routine—a pre-written UNIX routine that performs a common task, such as finding the square root of a number or allocating virtual memory. You can call any system routine from your program. The system routines used most often are UNIX Run-Time Library routines, utility routines, and system services.

10.1. The UNIX Procedure Calling and Condition Handling Standard

The UNIX Procedure Calling and Condition Handling Standard describes the concepts used by all UNIX languages for invoking routines and passing data between them. This standard specifies the PL/I parameter handling mechanism as well as defining the calling sequence, argument data types and descriptor formats, condition handling, and stack unwinding.

The following sections discuss the parameter handling mechanism in more detail.

10.2. Parameter-Passing Mechanisms

To pass data between routines that are not written in the same UNIX language, you must specify how you want that data to be represented and interpreted. You do this by specifying a parameter-passing mechanism. The general parameter-passing mechanisms and their keywords and abbreviations in PL/I are as follows:

- By reference—REFERENCE (REF)
- 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 established 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.

10.2.1. Passing Parameters by Reference

When you pass a parameter by reference, the PL/I 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 was not created.

You can pass an argument of any data type by reference. PL/I uses this mechanism by default 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 only needs to specify the data type of the parameter.

For example, the MEMSET runtime library routine requires the first argument passed by reference, and the second and third argument passed by value. You can declare this procedure as follows:

```
DECLARE MEMSET ENTRY (ANY, CHAR(1) VALUE,  
                     FIXED BIN(31) VALUE);
```

When this procedure is invoked, the first parameter can be any variable, the second parameter must be a character, and the third variable a numeric expression.

The data types in the parameter descriptors of all output arguments must match the data types of the written arguments.

10.2.1.1. 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 being passed by reference. The argument can be of any addressable data type known to PL/I. For example:

```
DECLARE MEMSET ENTRY (ANY, CHAR(1) VALUE,
                     FIXED BIN(31) VALUE);
```

The first parameter descriptor in the ENTRY attribute specifies that the argument can have any data type and is being passed by reference. 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 passes 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.

10.2.1.2. Dummy Arguments for Arguments Passed by Reference

When PL/I passes an argument by reference, it puts either the address of the actual argument or the address of a dummy argument in the argument list of the called procedure. PL/I creates 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 of the previous list, PL/I issues 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 issues an error message and does not create a dummy argument for the array or structure.

In creating a dummy argument, PL/I does the conversions listed in [Table 10.1, “Converted Data Types of Arguments Passed by Reference”](#):

Table 10.1. Converted Data Types of Arguments Passed by Reference

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

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

10.2.1.3. Using Pointer Values for Arguments Passed by Reference

When you pass an argument by reference, PL/I puts 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 MEMSET (POINTER VALUE, CHAR(1) VALUE,
               FIXED BIN(31) VALUE);
```

When MEMSET is invoked with the following statement at the procedure invocation, PL/I places the pointer value returned by the ADDR built-in function directly in the argument list.

```
CALL MEMSET(ADDR(EMPLOYEE_RECORD), ' ', 100);
```

10.2.1.4. Passing Arrays to a FORTRAN Routine by Reference

In FORTRAN, you must always pass arrays by reference; the array 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.

10.2.2. Passing Parameters by Value

When you pass a parameter by value, the PL/I compiler passes 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 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;
```

Arguments that can be passed by value are limited to the following data types, which can be expressed in 32 bits:

- 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) ALIGNED, where p is less than or equal to 32
- 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);
```

When the procedure is invoked, PL/I converts the written argument as needed to create a longword dummy argument.

10.2.2.1. Dummy Arguments for Arguments Passed by Value

For arguments to be passed by value, PL/I always creates a dummy argument directly in the argument list for the called procedure. If the parameter descriptor is specified with the ANY and VALUE attributes, PL/I creates dummy arguments the data types shown in Table 10.2, “Assignment of Data Type to Dummy Arguments”:

Table 10.2. Assignment of Data Type to Dummy Arguments

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

10.2.3. Special Parameter Attributes

PL/I provides the OPTIONS(VARIABLE) option to the ENTRY declaration, and the OPTIONAL and TRUNCATE parameter attributes.

- THE OPTIONS(VARIABLE) option indicates that the specified external procedure can be invoked with a variable number of arguments.

- The OPTIONAL attribute indicates that the 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.

The option and parameter attributes are described in the following sections.

10.2.3.1. OPTIONS(VARIABLE) Option

This option is provided for use in calling non-PL/I procedures and can only be used for that purpose. If you declare an external entry with OPTIONS(VARIABLE), you can omit an optional argument and let the compiler pass a zero for the argument. The called procedure usually provides a default value. You must, however, enter the correct number of commas to ensure that the argument list contains the correct number of longwords.

10.2.3.2. OPTIONAL Attribute

Some PL/I 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, 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 ASCTIM ENTRY (
    FIXED BINARY(15) OPTIONAL,
    CHARACTER(*),
    ANY OPTIONAL,
    BIT(1) ALIGNED VALUE OPTIONAL),
    TIME_STRING CHARACTER(24);
:
CALL ASCTIM(,TIME_STRING,,);
```

This call to the service ASCTIM specifies only the second argument; the argument list contains four arguments. When you specify a null argument, as in the previous example, PL/I always places a zero in the argument list passed to the called procedure.

10.2.3.3. TRUNCATE Attribute

Some routines may allow an argument list to be truncated in a specified location. For example:

```
DECLARE GET_INPUT ENTRY (ANY CHARACTER(*),  
    ANY CHARACTER(*) OPTIONAL TRUNCATE,  
    FIXED BINARY(15) OPTIONAL TRUNCATE);
```

This declaration routine (GET_INPUT) specifies that the argument list may vary in length. In fact, 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 GET_INPUT routine can be invoked with trailing arguments omitted:

```
CALL GET_INPUT(GET_STRING);
```

The variable GET_STRING is specified for the first argument. The other two arguments are not specified.

10.3. Calling Routines

The basic steps for calling routines are the same whether you are calling a routine written in PL/I, a routine written in some other UNIX language, a system service, a utility routine, or a UNIX Run-Time Library routine. The following sections outline the procedures for calling non-PL/I routines.

10.3.1. 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.

10.3.2. Declaring an External Routine and Its Arguments

To call a routine written in another UNIX 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

10.3.3. 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 (,) 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.

GNU Software Information

The following information pertains to the GNU readline and assembler software, used in the Endless PL/I for POSIX Systems software. The Free Software Foundation supplies the GNU Copyleft, which is provided verbatim in this appendix.

A.1. GNU Copyleft

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

A.1.1. Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

A.1.2. TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

A.1.3. How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

ONE LINE TO GIVE THE PROGRAM'S NAME AND AN IDEA OF WHAT IT DOES. Copyright (C) 19YY NAME OF AUTHOR This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19YY NAME OF AUTHOR Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'. This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w' and `show c'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program `Gnomovision' (which makes passes at compilers) written by James Hacker. SIGNATURE OF TY COON, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Run-Time Messages

This appendix describes the messages that the PL/I run-time library writes when a program ends because of a condition it does not handle.

B.1. Message Format

When a condition is raised and no ON-unit handles it, the default PL/I handler raises the ERROR condition (see [Section 9.5.1, “Default Handling for Main Procedures”](#)). If no ERROR ON-unit handles that either, the run-time library writes a report on standard error, deletes the program's temporary files, and ends the program with exit status 46.

The report has the following form. The lines in brackets appear only when they apply:

```
PL/I unhandled exception at 0xaddress:
condition-message
[status-message]
[IOERROR, I/O error on file 'file'.]
[FILENAME, File name: 'name'.]
[KEYWAS, Key was 'key'.]
[detail-message]
```

address

The address, in hexadecimal, of the instruction at which the condition was raised.

condition-message

The message for the condition that was raised.

status-message

The message for the status value of the error, the value that the ONCODE built-in function returns (see [Section 9.8.1, “ONCODE Built-In Function”](#)), when it differs from the condition.

IOERROR, FILENAME and KEYWAS

For an error in an I/O statement: the name of the file constant, the name of the file it was connected to, and, for a KEY condition, the key.

detail-message

For an I/O error, the message for the specific error, such as RNF, Record not found.

Most messages begin with an identification, such as RNF, followed by a comma and the text. The identifications are the names of the status values without their prefix: RNF is the message for RMS\$_RNF and NOKEY the message for PLI\$_NOKEY. [Section B.2, “Messages with Identifications”](#) lists these messages alphabetically by identification; [Section B.3, “Messages without Identifications”](#) lists the others.

In the message texts, *italic* words stand for values that the run-time library fills in.

B.2. Messages with Identifications

ACC, File access failed (must be opened NO_SHARE).

Explanation: An operation on a file failed because it needs access to the file that other users of the file prevent.

User Action: Open the file with the ENVIRONMENT option NO_SHARE, or run the program when no other program has the file open.

ACCVIO, Access violation.

Explanation: The program referred to memory that it is not allowed to read or write.

User Action: Look for a pointer or based variable that refers to storage that was never allocated or has been freed, a subscript outside the bounds of an array, or a mismatch between the parameters of a procedure and the arguments passed to it.

AREA, PL/I AREA condition.

Explanation: The AREA condition was raised, either by a SIGNAL AREA statement or by the run-time library.

User Action: If the run-time library raised the condition, the message that follows gives the reason.

AREA_ACTIVE, Operation attempted on an already active area.

Explanation: The run-time library found that an area it was about to operate on is already in use by another operation. This typically happens when an area is being changed both by a signal handler and by the code it interrupted, or when the area is in shared memory and is being changed by more than one process at once.

User Action: Change the program so that an area is not changed by two operations at the same time, or handle the condition with an ON-unit.

AREA_ASSIGN, Target area in area assignment is too small.

Explanation: In an area assignment, the target area is too small to hold the extent of the source area.

User Action: Make the target area larger, or add an ON-unit to handle the condition.

AREA_FORMAT, Operation attempted on an incorrectly formatted area.

Explanation: The run-time library found an area to be incorrectly formatted while allocating or freeing a variable in it, or while assigning from it.

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. Correct the program.

AREA_FREE, Error detected deallocating a variable in an area.

Explanation: A FREE statement for a variable in an area was not valid. For example, the variable being freed extends beyond the end of the area, extends into a free part of the area, or is already in a free part of the area. This normally happens when a variable of a different size from the one originally allocated is freed, or when a variable is freed more than once.

User Action: Correct the logic of the program.

AREA_FULL, No room for variable allocation in area at address *address*.

Explanation: An allocation in an area failed because the area does not have enough contiguous free space for the variable. The address is shown in hexadecimal.

User Action: Make the area larger, free variables in it that are no longer needed, or handle the AREA condition with an ON-unit.

AREA_INACTIVE, Attempt to unlock an area failed.

Explanation: The run-time library could not release its lock on an area after operating on it. This can happen when an area is assigned to while another operation on it is in progress, since an area assignment cannot be fully interlocked.

User Action: Change the program so that it does not assign to an area while another operation on the area is in progress.

AREA_SIZE, Impossible to allocate variable in area.

Explanation: An allocation in an area failed because the variable is larger than the whole area, so it could not be allocated even if the area were empty.

User Action: Make the area larger, or allocate the variable somewhere else.

AUTOINIT, Error in automatic initialization.

Explanation: An automatic variable declared with the INITIAL attribute could not be initialized. The cause can be a negative repetition factor in the initial list, or too few or too many values for an array. Automatic unconnected aggregates cannot be initialized.

User Action: Correct the initial list.

BADATTR, File attributes conflict with request.

Explanation: The attributes of a file conflict with the operation requested on it. This message indicates an error in the compiler or the run-time library.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

BASECTINIT, Error in BASED or CONTROLLED initialization.

Explanation: A BASED or CONTROLLED variable declared with the INITIAL attribute could not be initialized. The cause can be a negative repetition factor in the initial list, or too few or too many values for an array. BASED and CONTROLLED unconnected aggregates cannot be initialized.

User Action: Correct the initial list.

BIFSTAP0S, Starting position for a string built-in function is out of range.

Explanation: The third argument of an INDEX, SEARCH or VERIFY built-in function is beyond the range of the string. The run-time library always checks this; in-line code checks it only if the procedure was compiled with the - -check option.

User Action: Correct the third argument.

CHG, Invalid key change detected (CHANGES not set).

Explanation: A REWRITE statement would change the key of a record, and the index does not allow keys to change.

User Action: Do not change the key in a REWRITE. To change a key, delete the record and write it again with the new key.

CNVERR, Conversion error.

Explanation: An input or output value could not be converted, either by the default rules (for example, in list-directed stream I/O) or as the corresponding format item specifies (in edit-directed stream I/O).

User Action: Check the data and the logic of the program. For edit-directed I/O, check that each value is matched with the right format item.

CONAPPSUP, APPEND and SUPERSEDE conflict.

Explanation: The ENVIRONMENT options APPEND and SUPERSEDE were both given for the same file.

User Action: Decide whether to add records to the end of the existing file or to replace it with a new one, and remove the other option from the DECLARE or OPEN statement.

CONATTR, Conflicting attributes specified while opening file.

Explanation: A file was opened with attributes that conflict, either with each other or with the attributes in the file's declaration: for example, DIRECT and SEQUENTIAL, RECORD and STREAM, or more than one of INPUT, OUTPUT and UPDATE.

User Action: Decide the correct set of attributes for the file, and correct the file's declaration or the OPEN statement.

CONBLKIO, BLOCK_IO conflicts with other attributes or options.

Explanation: The ENVIRONMENT option BLOCK_IO was given for a file with one or more options that conflict with it.

User Action: Decide whether the file is to be opened for block I/O, and correct the file's declaration or OPEN statement.

CONDITION, PL/I CONDITION condition.

Explanation: A user-defined condition was signaled with a SIGNAL CONDITION statement and there is no ON-unit for it. The default action prints `CONDITION, PL/I CONDITION(value)` on standard error and continues the program.

User Action: No action is needed. To act on the condition, add an ON-unit for it.

CONENVOPT, DECLARED option conflicts with OPEN option.

Explanation: The value of an ENVIRONMENT option in the declaration of a file conflicts with the value given for it in the OPEN statement.

User Action: Decide which value is correct, and correct the declaration or the OPEN statement.

CONFIXLEN, FIXED_LENGTH_RECORDS conflicts with other attributes or options.

Explanation: The ENVIRONMENT option FIXED_LENGTH_RECORDS was given for a file with an option that conflicts with it.

User Action: See [Chapter 5, Options of the ENVIRONMENT Attribute](#) for the options that conflict, and correct the program.

CONPRINTCR, CARRIAGE_RETURN_FORMAT conflicts with PRINT attribute.

Explanation: A file with the PRINT attribute cannot have the ENVIRONMENT option CARRIAGE_RETURN_FORMAT: PL/I supplies the carriage control for a PRINT file.

User Action: Decide whether the file is to be a PRINT file, and correct the file's attributes.

CONPRTRFM, PRINTER_FORMAT conflicts with other attributes or options.

Explanation: The ENVIRONMENT option PRINTER_FORMAT conflicts with the CARRIAGE_RETURN_FORMAT option and with the PRINT and STREAM attributes.

User Action: Correct the file's attributes.

CONTROL C, Operation completed under CTRL/C.

Explanation: The program was interrupted from the keyboard.

User Action: No action is needed.

CONVERSION, PL/I CONVERSION condition.

Explanation: An invalid character was found while converting character data to another type. For example, the colon in '12:4' raises CONVERSION if the string is converted to a FIXED value.

User Action: Correct the data, or add an ON-unit for the CONVERSION condition. In the ON-unit, the ONSOURCE and ONCHAR built-in functions return the string and the character in error, and the pseudovariables of the same names can correct them.

CONVFILE, on file 'file'.

Explanation: Follows a CONVERSION message when the conversion was part of an I/O statement, and gives the name of the file constant.

User Action: None.

CREINDEX, Attempting to create an indexed file. Use the cif utility.

Explanation: An earlier run-time library could not create indexed files, and gave this message when a program tried. The current library creates an indexed file when a KEYED file with ENVIRONMENT(INDEXED) that does not exist is opened for OUTPUT or UPDATE.

User Action: None.

CUR, No current record.

Explanation: A DELETE or REWRITE statement without the KEY option was executed when there is no current record: for example, before any READ statement, or after the current record was deleted.

User Action: Read the record first, or give the KEY option.

CVTPICERR, Error in picture conversion.

Explanation: A value could not be edited as the picture specifies.

User Action: If the value is negative, check that the picture has a sign character. Check that the picture has enough digit positions for the value.

DEADLOCK, Deadlock detected.

Explanation: Two or more processes are each waiting for a record or file that another of them has locked.

User Action: Change the programs so that they lock records in the same order, or so that they do not wait for locked records.

DECOVF, Decimal overflow.

Explanation: The result of a fixed-point decimal operation or conversion is too large for its precision. This is reported as the FIXEDOVERFLOW condition.

User Action: Give the target a larger precision, or check the logic of the program. If the overflow is expected, handle FIXEDOVERFLOW with an ON-unit.

DEL, RFA-accessed record deleted.

Explanation: The record addressed by a RECORD_ID_FROM option has been deleted.

User Action: Check the record identification, or handle the error with an ON-unit.

DUP, Duplicate key detected (DUPLICATES not set).

Explanation: A WRITE or REWRITE statement would put a key in an index that does not allow duplicate keys, and a record with the same key already exists. For indexed files, only the first 20 bytes of a key are kept, so keys that differ only after byte 20 are duplicates.

User Action: Check the key, or handle the error with an ON-unit.

ENDSTRING, End of string encountered during GET STRING or PUT STRING.

Explanation: A GET STRING statement tried to read past the end of the source string, or a PUT STRING statement tried to write past the end of the target string. This happens most often when a GET STRING statement with the LIST option reads a string that does not end with a blank or a comma.

User Action: Check the length of the string and the data types in the GET or PUT list, and correct the program.

FILENAME, File name: 'name'.

Explanation: Follows the message for an I/O error, and gives the name of the operating-system file on which the operation failed.

User Action: Check that this is the file you expected: that the TITLE option and any defaults gave the right name.

FLK, File currently locked by another user.

Explanation: The file could not be opened because another process has it locked.

User Action: Run the program when the other process has closed the file, or open the file with sharing options that both programs allow.

FLTDIV, Floating/decimal divide by zero.

Explanation: A floating-point or decimal division had a divisor of zero. This is reported as the ZERODIVIDE condition.

User Action: Correct the logic of the program, or handle ZERODIVIDE with an ON-unit.

FLTOVF, Floating overflow.

Explanation: The result of a floating-point operation is too large to represent. This is reported as the OVERFLOW condition.

User Action: Correct the logic of the program, or handle OVERFLOW with an ON-unit.

FLTUND, Floating underflow.

Explanation: The result of a floating-point operation is too small to represent. This is reported as the UNDERFLOW condition.

User Action: Correct the logic of the program, or handle UNDERFLOW with an ON-unit.

FNF, File or object not found.

Explanation: The file named by the TITLE option, or by the file name, does not exist.

User Action: Check the file name and the current directory, or create the file first.

FORMATOVFL, Too many iteration factors or remote formats.

Explanation: A format list is too complex to be interpreted.

User Action: Simplify the format list of the stream I/O statement.

FPE, Floating point exception.

Explanation: A floating-point operation failed for a reason other than overflow, underflow or division by zero: for example, an invalid operation such as the square root of a negative number.

User Action: Correct the logic of the program.

FXCSIZ, FIXED_CONTROL_SIZE incorrect.

Explanation: The size of the variable in a FIXED_CONTROL_FROM or FIXED_CONTROL_TO option does not match the size of the file's fixed-control area.

User Action: Correct the size of the variable, or check that the right file is being used.

INCRETURN, RETURN statement is incompatible with ENTRY.

Explanation: A procedure does not have the RETURNS attribute, but an entry point in it has the RETURNS option and executed a RETURN statement.

User Action: Give the PROCEDURE statement the RETURNS option, or remove it from the ENTRY statement.

INTDIV, Integer divide by zero.

Explanation: An integer division had a divisor of zero. This is reported as the ZERODIVIDE condition.

User Action: Correct the logic of the program, or handle ZERODIVIDE with an ON-unit.

INTERNAL, PL/I compiler/run-time error. Please submit an SPR.

Explanation: An internal error occurred in the compiler or the run-time library.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

INTOVF, Integer overflow.

Explanation: The result of an integer operation is too large for its precision. This is reported as the FIXEDOVERFLOW condition.

User Action: Give the target a larger precision, or check the logic of the program. If the overflow is expected, handle FIXEDOVERFLOW with an ON-unit.

INVBKTSIZ, Invalid BUCKET_SIZE specified.

Explanation: The value of the ENVIRONMENT option BUCKET_SIZE is not in the range 0 through 32.

User Action: Correct the value.

INVBKLSIZ, Invalid BLOCK_SIZE specified.

Explanation: The value of the ENVIRONMENT option BLOCK_SIZE is not 0 and is not in the range 20 through 65532.

User Action: Correct the value.

INVDATYP, Invalid data type for record I/O.

Explanation: A variable in a record I/O statement does not have a computational data type, or it is an unconnected array or structure, an unaligned bit string, or an aggregate of unaligned bit strings.

User Action: Check that the right variable is named. If it is an aggregate, you may have to change its dimensions or structure so that it is connected.

INVDFNAM, Invalid DEFAULT_FILE_NAME.

Explanation: The value of the ENVIRONMENT option DEFAULT_FILE_NAME is not a valid character string, or is longer than 128 characters.

User Action: Correct the expression, or the variable it refers to.

INVEXTSIZ, Invalid EXTENSION_SIZE specified.

Explanation: The value of the ENVIRONMENT option EXTENSION_SIZE is not a valid integer in the range 0 through 65535.

User Action: Correct the value.

INVFMTPARM, Invalid format parameter specified.

Explanation: A value given for a format item is not a positive integer, or is outside the range for that format item. For example, a negative width for an A or B format item, or a width greater than 31 for an F format item.

User Action: Correct the value in the format item.

INVFORGKEY, Invalid file organization for KEYED access.

Explanation: The KEYED attribute was given for a file that cannot be accessed by key, such as a terminal or a pipe.

User Action: Check that the right file is being opened. If it is, remove the KEYED attribute from the file's attributes.

INVFXCISZ, Invalid FIXED_CONTROL_SIZE specified.

Explanation: The value of the ENVIRONMENT option FIXED_CONTROL_SIZE is not in the range 0 through 255.

User Action: Correct the value.

INVINDNUM, Invalid INDEX_NUMBER specified.

Explanation: The indexed file has no index with the number given in the INDEX_NUMBER option. Only the primary index, number 0, is supported.

User Action: Remove the INDEX_NUMBER option, or give it the value 0.

INVMAXREC, Invalid MAXIMUM_RECORD_SIZE specified.

Explanation: The value of the ENVIRONMENT option MAXIMUM_RECORD_SIZE is not in the range 0 through 32767.

User Action: Correct the value.

INVMLTBLK, Invalid MULTIBLOCK_COUNT specified.

Explanation: The value of the ENVIRONMENT option MULTIBLOCK_COUNT is not a valid integer in the range 0 through 127.

User Action: Correct the value.

INVMLTBUF, Invalid MULTIBUFFER_COUNT specified.

Explanation: The value of the ENVIRONMENT option MULTIBUFFER_COUNT is not a valid integer in the range -128 through 127.

User Action: Correct the value.

INVOWNGRP, Invalid OWNER_GROUP specified.

Explanation: The value of the ENVIRONMENT option OWNER_GROUP is not a valid integer in the range 0 through 16383.

User Action: Correct the value.

INVOWNMEM, Invalid OWNER_MEMBER specified.

Explanation: The value of the ENVIRONMENT option OWNER_MEMBER is not a valid integer in the range 0 through 65535.

User Action: Correct the value.

INVPROT, Invalid protection string specified.

Explanation: The value of one of the ENVIRONMENT options GROUP_PROTECTION, OWNER_PROTECTION, SYSTEM_PROTECTION or WORLD_PROTECTION is not a valid string, is longer than four characters, or contains characters other than R, W, E and D (in upper or lower case).

User Action: Correct the value.

INVRADIX, An invalid radix was specified.

Explanation: The radix given to the ENCODE or DECODE built-in function is not in the range 2 through 16.

User Action: Give a radix in the range 2 through 16.

INVRTVPTR, Invalid RETRIEVAL_POINTERS specified.

Explanation: The value of the ENVIRONMENT option RETRIEVAL_POINTERS is not a valid integer in the range -1 through 127.

User Action: Correct the value.

INVSkip, Invalid value for SKIP option specified.

Explanation: The value of a SKIP option is zero on input, or negative.

User Action: Correct the value, or the variable it comes from.

INVSTRFMT, Invalid format item for STRING I/O.

Explanation: One of the format items COLUMN, SKIP, LINE, PAGE or TAB was used in a GET STRING or PUT STRING statement, where they are not allowed.

User Action: Correct the format list.

INVSTRING, Invalid character encountered in string.

Explanation: The string given to the DECODE built-in function contains characters that are not valid.

User Action: Remove the invalid characters.

INVTIME, Invalid timeout value specified.

Explanation: The timeout given for a record lock is not valid. The number of seconds to wait must be less than 256.

User Action: Correct the timeout value.

INV_KEY, Invalid KEY data type.

Explanation: The data type of a key in an indexed file is not one that PL/I recognizes.

User Action: Check that the file has not been damaged. If it has, go back to an earlier copy.

IOERROR, I/O error on file '*file*'.

Explanation: An error occurred in an I/O operation on the file. The messages that follow give the cause.

User Action: See the messages that follow.

IOP, Operation invalid for file organization.

Explanation: The I/O operation is not valid for the organization of the file: for example, a keyed operation on a file that is not indexed or relative.

User Action: Check the I/O statement and the file's attributes and ENVIRONMENT options.

KEY, Invalid record number key or key value.

Explanation: The key or record number given in an I/O statement is not valid for the file: for example, a record number of zero or less.

User Action: Correct the key, or handle the KEY condition with an ON-unit.

KEYWAS, Key was 'key'.

Explanation: Follows the message for a KEY condition on a file, and shows the key that caused it.

User Action: Check the key, or handle the KEY condition with an ON-unit.

KRF, Invalid key-of-reference.

Explanation: The index named or numbered in the I/O operation does not exist in the file.

User Action: Correct the index number, or check that the right file is being used.

LABELRANGE, Uninitialized label subscript used.

Explanation: A GOTO statement refers to a subscripted label through a variable that has no valid value.

User Action: Correct the logic of the program so that the variable has a valid value.

LINESIZE, Invalid LINESIZE specified.

Explanation: The value of the LINESIZE option is not a positive integer, or exceeds the limit of 32767.

User Action: Correct the value.

LINOVRFLO, Line number overflow.

Explanation: The number of lines on a page of a stream file exceeds the limit of 32767.

User Action: Correct the program, or handle the condition with an ANYCONDITION or VAXCONDITION ON-unit.

NEF, Not positioned to EOF.

Explanation: A sequential WRITE was attempted on a file that is not positioned at its end.

User Action: Open the file with the ENVIRONMENT option APPEND to add records to the end, or position the file before writing.

NOCURREC, No current record.

Explanation: A DELETE or REWRITE statement without the KEY option was executed for an UPDATE file, but there is no current record.

User Action: Read the record first, or give the KEY option.

NOFROM, No FROM specified or buffer not allocated.

Explanation: A REWRITE statement without the FROM option is valid only if a previous READ statement for the file used the SET option.

User Action: Add the FROM option to the REWRITE statement, or use the SET option on the READ statement.

NOKEY, No KEY or KEYFROM specified.

Explanation: A keyed I/O statement for a DIRECT file must have a KEY or KEYFROM option.

User Action: Add the option. If the file is to be read or written in sequence, give it the SEQUENTIAL attribute.

NOWSELECT, No WHEN clause selected and no OTHERWISE specified.

Explanation: None of the WHEN clauses of a SELECT group was selected, and the group has no OTHERWISE clause.

User Action: Add an OTHERWISE clause to the SELECT group.

NOSHARE, SHARED_READ or SHARED_WRITE conflict with NO_SHARE.

Explanation: The ENVIRONMENT options SHARED_READ and SHARED_WRITE allow a file to be shared, and NO_SHARE forbids all sharing; they cannot be given together.

User Action: Decide whether the file is to be shared, and remove the options that do not apply.

NOTIMPL, Language feature not implemented for PL/I for ULTRIX/RISC.

Explanation: The program uses a feature that this version of the run-time library does not implement: for example, a nonzero INDEX_NUMBER, or one of the IBM dialect ENVIRONMENT options such as KEYLENGTH, KEYLOC or GENKEY.

User Action: Remove the feature from the program.

NOTINDEXED, Requested operation requires an INDEXED file.

Explanation: A keyed I/O statement requests an operation that is valid only for an indexed file.

User Action: Check, from the FILENAME message, that the operation was applied to the right file. If the file should be indexed, declare it with ENVIRONMENT(INDEXED).

NOTINPUT, Attempting to GET from an OUTPUT or UPDATE file.

Explanation: A GET statement is not valid for a file opened with the OUTPUT or UPDATE attribute.

User Action: Correct the file's attributes.

NOTKEYD, Not a KEYED file.

Explanation: A KEY or KEYFROM option was given in a record I/O statement for a file without the KEYED attribute.

User Action: If the file is to be accessed by key, give it the KEYED attribute in its DECLARE or OPEN statement.

NOTOUT, Attempting to PUT to an INPUT or UPDATE file.

Explanation: A PUT statement is not valid for a file opened with the INPUT or UPDATE attribute.

User Action: Correct the file's attributes.

NOTPRINT, PAGE or LINE specified for non-PRINT file.

Explanation: The PAGE and LINE options and format items are valid only for a file with the PRINT attribute.

User Action: If the file is a stream output file meant for printing, give it the PRINT attribute.

NOTREC, Not a RECORD file.

Explanation: A record I/O statement (READ, WRITE, DELETE or REWRITE) was executed for a file with the STREAM attribute.

User Action: Correct the file's attributes, or use stream I/O statements for the file.

NOTRELSQL, Not a RELATIVE or SEQUENTIAL file

Explanation: The option to lock a nonexistent record can be used only with relative or sequential files.

User Action: Remove the LOCK_NONEXISTENT option, or use a different file organization.

NOTSQL, Not a SEQUENTIAL file.

Explanation: A READ or WRITE statement without a KEY or KEYFROM option was executed for a file with the DIRECT attribute.

User Action: Decide whether the file is to be accessed in sequence or by key, and correct the statement or the attributes.

NOTSTREAM, Stream I/O attempted on RECORD file.

Explanation: A GET or PUT statement was executed for a file with the RECORD attribute.

User Action: Correct the file's attributes, remembering that some attributes, such as KEYED, imply RECORD.

NOTUPDATE, Attempting to REWRITE or DELETE an INPUT or OUTPUT file.

Explanation: The REWRITE and DELETE statements are valid only for a file with the UPDATE attribute.

User Action: Correct the file's attributes.

NOVIRMEM, Virtual memory overflow.

Explanation: The run-time library could not allocate memory, for example for an I/O buffer.

User Action: Reduce the memory the program uses, or raise the limit on the memory a process may use.

ONCNVPOS, The erroneous character is at position *position*.

Explanation: Follows a CONVERSION message, and gives the position of the character in error in the source string.

User Action: Use the position to correct the data.

ONSNOTMOD, ONSOURCE value not modified.

Explanation: A CONVERSION ON-unit ended normally, which retries the conversion, but it did not change the ONSOURCE value. To prevent an endless loop, the ERROR condition was raised instead.

User Action: In the ON-unit, correct the source with the ONSOURCE or ONCHAR pseudovisible, or leave the ON-unit with a GOTO statement.

ONSOURCE, The conversion source is 'string'.

Explanation: Follows a CONVERSION message, and shows the string being converted.

User Action: None.

OPEN, Open failure.

Explanation: An I/O statement opened a file implicitly, but the file could not be opened. The UNDEFINEDFILE condition was raised, and its ON-unit did not open the file.

User Action: Write an UNDEFINEDFILE ON-unit for the file that opens it, or open the file explicitly.

PAGESIZE, Invalid PAGESIZE specified.

Explanation: The value of the PAGESIZE option is not a positive integer, or exceeds the limit of 32767.

User Action: Correct the value.

PAGOVRFLO, Page number overflow.

Explanation: The number of pages in a stream file exceeds the limit of 32767.

User Action: Correct the program, or handle the condition with an ANYCONDITION or VAXCONDITION ON-unit.

POS, Invalid key position (greater than MRS).

Explanation: The key position given for an index lies outside the maximum record size of the file.

User Action: Correct the key position, or the maximum record size.

PROMPTOBIG, PROMPT option too long. Must be less than 254 characters.

Explanation: The string in the PROMPT option of a GET statement is longer than 253 characters.

User Action: Shorten the prompt.

PRV, Insufficient privilege or file protection violation.

Explanation: The file could not be opened or created because its permissions, or those of its directory, do not allow it.

User Action: Check the permissions of the file and directory, or run the program as a user who has access.

READOUT, Attempting to READ from an OUTPUT file.

Explanation: A READ statement is not valid for a file opened with the OUTPUT attribute. To read a file that has just been written, close it and open it again with the INPUT attribute.

User Action: Correct the program.

RECID, File not open for RECORD_ID_ACCESS.

Explanation: The RECORD_ID_TO and RECORD_ID_FROM options are valid only if the file was opened with the ENVIRONMENT option RECORD_ID_ACCESS.

User Action: Add RECORD_ID_ACCESS to the ENVIRONMENT options of the file.

RECIDKEY, RECORD_ID conflicts with KEY or KEYFROM.

Explanation: A record I/O statement cannot have both the RECORD_ID_FROM option and the KEY or KEYFROM option.

User Action: Correct the statement.

RECLCOPT, Conflicting record-locking options requested.

Explanation: A READ statement gives record-locking options that conflict with each other.

User Action: Correct the options.

RECORD, Record length does not match target length.

Explanation: A record read by a READ statement is not the same length as the fixed-length variable it is read into, or an area is too small to hold the extent of the area being read into it.

User Action: Check that the variable is the right length for the records in the file.

REX, Record already exists.

Explanation: A WRITE statement tried to write a record to a relative or indexed file at a position, or with a key, where a record already exists.

User Action: Use REWRITE to replace the record, or handle the error with an ON-unit.

RFA, Invalid record's file address (RFA).

Explanation: The record identification given in a RECORD_ID_FROM option is not valid for the file.

User Action: Use only record identifications returned by a RECORD_ID_TO option for the same file.

RLK, Target record currently locked by another stream.

Explanation: The record is locked by another process or another open of the same file.

User Action: Try again later, or use the WAIT_FOR_RECORD option.

RNF, Record not found.

Explanation: No record has the key or record number given in the I/O statement. For indexed files, only the first 20 bytes of the key are compared.

User Action: Check the key, or handle the error with an ON-unit.

RNL, Record not locked.

Explanation: The program tried to release a lock on a record that is not locked.

User Action: Correct the program.

RSZ, Invalid record size.

Explanation: The size of a record being written is not valid for the file: for example, larger than its maximum record size, or a different size from the fixed records of the file.

User Action: Check the length of the variable written, and the record size of the file.

SEGV, Segment violation.

Explanation: The program referred to memory outside its address space.

User Action: Look for a pointer or based variable that refers to storage that was never allocated or has been freed, a subscript outside the bounds of an array, or a mismatch between the parameters of a procedure and the arguments passed to it.

STORAGE, PL/I STORAGE condition.

Explanation: Memory for an ALLOCATE statement could not be obtained.

User Action: Reduce the memory the program uses, or free variables that are no longer needed.

STROVFL, Stream item too big. Must be less than 1000 characters.

Explanation: The run-time library cannot process a stream I/O item longer than 1000 characters.

User Action: Use a smaller field width, or more than one stream I/O statement.

STRANGE, PL/I STRINGRANGE condition.

Explanation: The arguments of a SUBSTR built-in function or pseudovisible, or the third argument of an INDEX, SEARCH or VERIFY built-in function, are outside the range of the string. The run-time library always checks this; in-line code checks it only if the procedure was compiled with the --check option.

User Action: Correct the reference.

STRSIZE, PL/I STRINGSIZE condition.

Explanation: A string was assigned to a target too short to hold it, and the STRINGSIZE condition is enabled.

User Action: Correct the length of the target, or the value assigned to it.

SUBRANGE, Subscript range check error.

Explanation: A subscript is outside the bounds of its array. This is checked only if the procedure was compiled with the --check option.

User Action: Correct the reference.

SUBRANGEn, Subscript n range check error.

Explanation: Subscript *n* (1 through 8) of an array reference is outside the bounds of that dimension. This is checked only if the procedure was compiled with the --check option.

User Action: Correct the reference.

SUBRG, PL/I SUBSCRIPTRANGE condition.

Explanation: A subscript is outside the bounds of its array. This is checked only if the procedure was compiled with the --check option.

User Action: Correct the reference.

SUBSTR2, Operand 2 of a SUBSTR is out of range.

Explanation: The second argument of a SUBSTR built-in function or pseudovvariable is outside the range of the string. This is checked only if the procedure was compiled with the --check option.

User Action: Correct the reference.

SUBSTR3, Operand 3 of a SUBSTR is out of range.

Explanation: The third argument of a SUBSTR built-in function or pseudovvariable is outside the range of the string. This is checked only if the procedure was compiled with the --check option.

User Action: Correct the reference.

TITLE, Invalid TITLE specified.

Explanation: The string in the TITLE option is longer than 128 characters.

User Action: Use a shorter file name.

TMO, Timeout period expired (WAIT_FOR_RECORD).

Explanation: A locked record did not become free within the time allowed.

User Action: Try again later, or allow a longer time.

TRANSMIT, PL/I compiler/run-time error. Please submit an SPR.

Explanation: The TRANSMIT condition was raised. This message indicates an error in the compiler or the run-time library.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

TRE, Error in index tree (file corrupted).

Explanation: The index of an indexed file is damaged.

User Action: Go back to an earlier copy of the file.

WRITEIN, Attempting to WRITE to an INPUT file.

Explanation: A WRITE statement is not valid for a file opened with the INPUT attribute. To write to a file that has just been read, close it and open it again with the UPDATE attribute, or with the OUTPUT attribute and ENVIRONMENT(APPEND).

User Action: Correct the program.

B.3. Messages without Identifications

The messages for most of the PL/I conditions, and a few others, have no identification. They are listed here in alphabetical order.

Invalid file control block or uninitialized FILE VARIABLE.

Explanation: An I/O statement refers to a file variable that has no value, or to a file whose control block has been damaged.

User Action: Check that every file variable is assigned a file constant before it is used.

PL/I ANYCONDITION condition.

Explanation: ANYCONDITION was signaled by a SIGNAL statement, and there is no ON-unit for it.

User Action: Add an ON-unit for ANYCONDITION, or remove the SIGNAL statement.

PL/I compiler/run-time error. Please submit an SPR.

Explanation: An internal error occurred in the compiler or the run-time library. In this version, the same text is also printed for the SIZE and NAME conditions.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

PL/I ENDFILE condition on file '*file*'.

Explanation: A READ or GET statement tried to read past the end of the file, and there is no ENDFILE ON-unit for the file.

User Action: Add an ENDFILE ON-unit for the file.

PL/I ENDPAGE condition on file '*file*'.

Explanation: A PUT statement took the line number past the page size of a PRINT file, and there is no ENDPAGE ON-unit for the file.

User Action: If the file is a terminal, the null ON-unit ON ENDPAGE (SYSPRINT) ; makes PL/I ignore the condition. For other print files, write an ENDPAGE ON-unit to take the action you need, such as printing a heading.

PL/I ERROR condition.

Explanation: The ERROR condition was raised, and there is no ERROR ON-unit. It is usually followed by messages that give the error.

User Action: See the messages that follow.

PL/I FIXEDOVERFLOW condition.

Explanation: The result of a fixed-point operation is too large for its precision, and there is no FIXEDOVERFLOW ON-unit.

User Action: Give the target a larger precision, or check the logic of the program. If the overflow is expected, add a FIXEDOVERFLOW ON-unit.

PL/I internal FAB condition.

Explanation: An internal error occurred in the run-time library while opening or closing a file.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

PL/I internal RAB condition.

Explanation: An internal error occurred in the run-time library during a record operation.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

PL/I KEY condition on file 'file'.

Explanation: An error occurred with the key in a keyed I/O statement, and there is no KEY ON-unit for the file. The messages that follow give the error and the key.

User Action: Check that the key is correct and can be converted to the type of the key of the file, and that the right file is being used. If appropriate, add a KEY ON-unit.

PL/I Program FINISH condition.

Explanation: Identifies the FINISH condition. When there is no FINISH ON-unit, FINISH ends the program normally.

User Action: None.

PL/I run-time error: File ident not valid. Please submit an SPR.

Explanation: An internal error occurred in the run-time library.

User Action: Report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

PL/I UNDEFINEDFILE condition on file 'file'.

Explanation: The file could not be opened, and there is no UNDEFINEDFILE ON-unit for it. The messages that follow give the reason.

User Action: See the messages that follow. Check, from the FILENAME message, that the file name is the one you expected.

PL/I ZERODIVIDE condition.

Explanation: A division had a divisor of zero, and there is no ZERODIVIDE ON-unit.

User Action: Correct the logic of the program. If practical, add a ZERODIVIDE ON-unit.

Run-time library does not support indexed files. Re-link with -isam option.

Explanation: A program that uses indexed files was linked without indexed file support.

User Action: Link the program with indexed file support.

Unknown condition: *decimal* (0*hexadecimal*).

Explanation: The condition value has no message. The value is shown in decimal and hexadecimal.

User Action: For a condition value from a SIGNAL statement or a system service, look up the value. Otherwise, report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

VAXCONDITION condition: *value*.

Explanation: A VAXCONDITION was signaled with the given value, and there is no ON-unit for that value.

User Action: Check that the value in the SIGNAL statement matches the value in the ON statement.

B.4. Indexed File Messages

An error in an operation on an indexed file (a file declared with `ENVIRONMENT(INDEXED)`) raises the `ERROR` condition. Most errors are reported with the message of the corresponding status value: for example, a key that is not in the file gives `RNF`, `Record not found`, and a duplicate key gives `DUP`, `Duplicate key detected` (`DUPLICATES not set`). The errors in [Table B.1, “Indexed File Messages”](#) have no such status value, and are reported with a message of their own, without an identification. Apart from `File in use` and `Record is in use`, which arise when other processes are using the file, they usually mean that the program and the file do not match, or that the file is damaged; if neither is the case, report the problem to Endless Software Solutions.

Table B.1. Indexed File Messages

Message	Meaning
Beginning of file	A sequential operation tried to move before the first record of the file.
Deleted Record	The record has been deleted.
Duplicate Index defined	An index with the same name already exists in the file.
Explicit key required	The operation needs a key, and none was given.
File in use	Another process has the file locked, so it cannot be opened or locked.
Index not found	The file has no index with the name or number given.
Invalid index	No index was named when the indexed file was opened.
Invalid Keyed Relation	The relation requested for a keyed read is not valid.
Invalid Record Number	The record number is zero, negative, or beyond the last record of the file.
Item Index	The operation cannot be used with this kind of index.
Key is not the same	The key given in a <code>WRITE</code> statement is not the key contained in the record.
Long Record	The record is longer than the variable it is being read into.
Not Item index	The operation can be used only with an index of another kind.
Not locked to this process	The program tried to unlock a file that another process has locked.
Not open for output	The program tried to lock a file for writing that was opened for <code>INPUT</code> .
Record is in use	Another process has the record locked.
Record too long	The record is longer than the maximum record length of the file.
Record too short	The record is shorter than the minimum record length of the file, or too short to contain its key.

B.5. Panic Messages

If the run-time library's own condition handling fails, it writes one of the following messages on standard error and ends the program at once with exit status 77, without running any `ON`-units:

- Panic: Invalid PLI exception code, aborting

- Panic: PLI Signal Stack Overflow, aborting
- Panic: PLI Signal Stack Underflow, aborting
- Panic: PLI Signal error, aborting

A stack overflow can mean that ON-units are raising conditions without end. Otherwise, report the problem to Endless Software Solutions, with as much information as possible about the circumstances in which it occurred.

Compiler Messages

This appendix lists the diagnostic messages of the PL/I compiler, in order of message number.

Each message is written to standard output, and to the listing file if there is one, in the following form (see Section 2.2.5, “Compiler Error Messages”):

```
*****ERROR nn SEVERITY s on line line-no in file filename *****
      message-text
```

For a message of severity 1, the first word is WARNING instead of ERROR. The text is broken into lines of about 60 characters, each indented three spaces. Where a message refers to something in the program, such as the name of a variable, it is shown here as *name*.

The severities are:

1 (Informational)

The program is valid, but uses a construct that may not be what you intended or may not be portable.

2 (Warning)

The compiler found a problem and has corrected it; check that its correction is what you intended. An object module is produced.

3 (Error)

The program is in error. The compiler continues, to find any further errors, but produces no object module.

4 (Fatal)

The compiler cannot continue, and stops at once.

After 100 messages of severity 3 or 4, the compiler reports message 71 and stops. The numbers are not consecutive: numbers that this compiler does not use are left out.

1 The source text contains extraneous data. Check for excess END statements, unbalanced quotes, and unbalanced /* */.

Severity: 3 (Error)

Explanation: 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.

2 Insufficient END statements. An additional END statement has been supplied.

Severity: 2 (Warning)

Explanation: The compiler inserted an END statement in the file.

User Action: The listing file, if any, shows where the compiler inserted the END statement. Check that it is in the right place, and add the END statement to the source program.

3 This is an unrecognizable statement.

Severity: 3 (Error)

Explanation: 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.

4 Invalid range specification in this DEFAULT statement.

Severity: 3 (Error)

Explanation: A DEFAULT statement has an invalid range.

User Action: Correct the values given for the range.

5 "name" is an unrecognizable attribute keyword.

Severity: 3 (Error)

Explanation: The attribute is not a keyword of the dialect being compiled.

User Action: Remove or correct the attribute keyword, or compile with the dialect that has it (the -D option).

6 "name" is an unrecognizable format-item.

Severity: 3 (Error)

Explanation: A format list in a GET EDIT or PUT EDIT statement contains a character that is not a valid PL/I format item.

User Action: Verify the syntax of the format list and ensure that the format item is a valid PL/I format item.

7 "name" is a label prefix but has previously been declared.

Severity: 3 (Error)

Explanation: 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.

8 "name" is a subscripted label prefix but has previously been declared with a different data type or a different number of dimensions.

Severity: 3 (Error)

Explanation: 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.

9 This statement contains a label prefix that has appeared on a previous statement in the same block.

Severity: 3 (Error)

Explanation: 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.

10 An expression or reference contained within this DECLARE or DEFAULT statement is excessively complex and has caused the compiler's expression buffer to overflow. Reduce the complexity and recompile.

Severity: 4 (Fatal)

Explanation: The compiler cannot interpret the statement.

User Action: Rewrite the declaration using two or more statements.

11 A colon cannot follow a WHEN or OTHERWISE component of a SELECT statement. The colon has been ignored.

Severity: 2 (Warning)

User Action: No action is needed, but you may want to remove the colon to avoid the message.

12 If any entry of a procedure has a RETURNS option, all entries must have a RETURNS option. A procedure cannot be both a subroutine and a function.

Severity: 3 (Error)

Explanation: The main procedure entry and alternate entries must all be declared similarly with respect to their usage as either a subroutine or a function.

User Action: Change the entry declarations to match the main entry declaration.

13 A non-ascii character has been found on this line.

Severity: 3 (Error)

Explanation: The compiler encountered an invalid character in the source line.

User Action: Remove the character.

14 "name" is a parameter but has been declared with a storage class or as a label.

Severity: 3 (Error)

Explanation: 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.

15 "name" as used in this statement conflicts with its usage in a previous statement within the same block.

Severity: 3 (Error)

Explanation: The name is used in a way that conflicts with an earlier use of it in the same block.

User Action: Examine both the specified statement and the other usages in the block and determine which is correct. Change the incorrect usages.

16 A label prefix cannot appear on a statement used as an on-unit.

Severity: 3 (Error)

Explanation: An ON-unit action statement contains a label.

User Action: Remove the label from the statement.

17 "name" is an apparent structure member, but does not immediately follow a variable with a level-number.

Severity: 3 (Error)

Explanation: 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.

19 The closure label in this statement does not match the label prefix of any containing DO, BEGIN, or PROCEDURE statement. This statement has closed the external procedure and will probably cause false errors to be reported.

Severity: 3 (Error)

Explanation: Multiple closure is not permitted in PL/I. 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.

20 An IF, ON, DO, RETURN, LEAVE, or SELECT statement cannot be used as an on-unit.

Severity: 3 (Error)

Explanation: 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.

21 This statement contains a duplicate declaration of "name".

Severity: 3 (Error)

Explanation: 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.

22 The total number of declarations, parameter descriptors, and returns descriptors in this DECLARE statement exceeds the implementation's limit of 8192.

Severity: 4 (Fatal)

Explanation: 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.

23 This statement contains duplicate WHILE options.

Severity: 3 (Error)

Explanation: The same WHILE option is used more than once in a single statement.

User Action: Remove one of the duplicated options.

24 A SYSTEM option appears as a statement.

Severity: 3 (Error)

User Action: Correct the statement.

25 Attributes declared for "name" conflict with factored attributes.

Severity: 3 (Error)

Explanation: 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.

26 This statement contains unbalanced parentheses.

Severity: 3 (Error)

Explanation: A statement has different numbers of left and right parentheses.

User Action: Find where a parenthesis is missing or extra, and correct the statement.

27 Invalid syntax in this SELECT statement.

Severity: 3 (Error)

User Action: Correct the SELECT statement.

28 Invalid end of text. Check for unbalanced quotes or unbalanced comments.

Severity: 3 (Error)

Explanation: The end of the source file was reached inside a string constant or a comment.

User Action: Find the string constant or comment that is not closed, and close it.

29 An identifier contains more than 32 characters. Only the first 32 characters will be used.

Severity: 3 (Error)

Explanation: An identifier is longer than 32 characters.

User Action: Shorten the identifier.

30 A FORMAT, ENTRY, PROCEDURE, END, DECLARE, or DEFAULT statement appears in a context that requires an executable single statement, do-group, or begin-block.

Severity: 3 (Error)

Explanation: 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.

31 A factored level cannot be applied to "name".

Severity: 3 (Error)

Explanation: 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.

32 Previous errors prevent continued compilation. If no previous errors were diagnosed, contact compiler maintenance personnel.

Severity: 4 (Fatal)

Explanation: Previous errors have made continued compilation impossible.

User Action: Correct any error in the program and recompile.

33 Invalid syntax in attribute list.

Severity: 3 (Error)

Explanation: 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.

34 The GENERIC attribute cannot be nested.

Severity: 3 (Error)

Explanation: A GENERIC attribute appears inside the GENERIC attribute of another name.

User Action: Declare each generic name separately.

35 PROCEDURE, ENTRY, and FORMAT statements must have a label.

Severity: 3 (Error)

Explanation: The indicated statement is not labeled.

User Action: Place a label on the statement that caused the error.

36 "name" is an unrecognizable attribute.

Severity: 3 (Error)

Explanation: A declaration contains a keyword that is not a valid attribute.

User Action: Check the statement for a misspelling, and check that commas, parentheses and spaces appear where required.

37 An END statement has been supplied to close a do-group, begin-block, or procedure.

Severity: 1 (Informational)

Explanation: 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.

38 The scale-factor of a constant exceeds the implementation's limits of $-128 \leq q \leq 127$.

Severity: 3 (Error)

Explanation: You specified an incorrect scale factor.

User Action: Correct the code.

39 DEFAULT NONE; will produce invalid declarations and ERROR 32 unless every declared name has been explicitly declared with a data type, storage class, and alignment attributes.

Severity: 1 (Informational)

User Action: None, unless the compiler also reports other errors.

40 This LEAVE statement is either not contained within a do-group, or it attempts to transfer control out of the current procedure or begin-block.

Severity: 3 (Error)

Explanation: 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.

41 This statement contains an ambiguous reference to "name".

Severity: 3 (Error)

Explanation: 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.

46 "*name*" is declared with conflicting or duplicate attributes.

Severity: 3 (Error)

Explanation: The declaration gives the name attributes that conflict, such as FIXED and FLOAT, or gives the same attribute twice.

User Action: Correct the declaration.

49 An expected *name* value was not found. One of the values in this statement has a data type which is inconsistent with the context in which the value is used.

Severity: 3 (Error)

Explanation: 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.

50 The precision specified for "*name*" exceeds the implementation's limit of FIXED BINARY(31). The maximum precision of 31 has been supplied. Use FIXED DECIMAL for larger values.

Severity: 2 (Warning)

Explanation: 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.

51 The precision specified for "*name*" exceeds the implementation's limit of FIXED DECIMAL(31). The maximum precision of 31 has been supplied.

Severity: 2 (Warning)

Explanation: 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.

52 The scale-factor specified for "*name*" exceeds the implementation's limit of $-14 \leq q \leq 14$. The scale-factor has been set to zero.

Severity: 2 (Warning)

Explanation: A scale is specified for a variable that is not in its declared range.

User Action: Specify a scale factor in the allowed range.

53 The precision specified for "*name*" exceeds the implementation's limit of FLOAT BINARY(53). The maximum precision of 53 has been supplied.

Severity: 2 (Warning)

Explanation: 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.

54 The precision specified for "*name*" exceeds the implementation's limit of FLOAT DECIMAL(15). The maximum precision of 15 has been supplied.

Severity: 2 (Warning)

Explanation: 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.

56 More than 8 dimensions have been specified in a DIMENSION attribute.

Severity: 3 (Error)

Explanation: 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.

57 PASS2 has overflowed its stack. Reduce the complexity of this statement and recompile.

Severity: 4 (Fatal)

User Action: Reduce the complexity of this statement and recompile.

58 A descriptor is an apparent structure member, but does not immediately follow a descriptor with a level-number.

Severity: 3 (Error)

Explanation: 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.

59 "*name*" is declared with an * extent but is not a parameter or a descriptor. If this is intended to be a CONTROLLED variable use the compile option for IBM compatibility.

Severity: 3 (Error)

Explanation: 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.

60 "*name*" is declared with nonconstant extents but is not an automatic, based, controlled, or defined variable.

Severity: 3 (Error)

Explanation: 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.

61 An expected *name* 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.

Severity: 3 (Error)

Explanation: 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.

62 Possible compiler error. An invalid code has been given to the convert subroutine of PASS2. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 3 (Error)

Explanation: An internal compiler error has been detected.

User Action: If this message persists, report the problem to Endless Software Solutions.

63 The length of a name or constant exceeds the implementation's limit of 4000 characters. Check to see if all string constants are properly quoted with ' and that any contained 's occur in pairs. Also check for unbalanced /* */.

Severity: 4 (Fatal)

Explanation: A name or constant is longer than 4000 characters. This usually means that a string constant or a comment has not been closed.

User Action: Close the string constant or comment at the right place.

64 A built-in function has been referenced with the wrong number of arguments.

Severity: 3 (Error)

Explanation: 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.

66 The depth of nesting of a structure exceeds the implementation's limit of 16.

Severity: 4 (Fatal)

Explanation: 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.

67 A statement exceeds the implementation's limit of 4096 tokens.

Severity: 4 (Fatal)

Explanation: The statement contains more than 4096 constants, identifiers, and punctuation symbols, including operators.

User Action: Simplify the statement.

68 Invalid syntax in an arithmetic constant.

Severity: 3 (Error)

Explanation: 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.

69 An arithmetic constant must be separated from the following symbol by a delimiter. A blank delimiter has been supplied.

Severity: 2 (Warning)

Explanation: 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.

70 A symbol begins with _. The _ has been ignored.

Severity: 3 (Error)

Explanation: The compiler encountered a symbol beginning with an underscore.

User Action: None. You may want to remove the underscore from the symbol name to avoid getting this message in the future.

71 The total number of errors exceeds the implementation's limit of 100.

Severity: 4 (Fatal)

Explanation: Compilation has stopped due to the number of errors encountered.

User Action: Correct the reported errors and recompile.

72 The second argument of the round built-in function must be an optionally signed decimal integer constant.

Severity: 3 (Error)

Explanation: A reference to the ROUND built-in function specifies an invalid argument.

User Action: Correct the argument in the reference to the ROUND built-in function.

73 The R-format must reference a format-value.

Severity: 3 (Error)

Explanation: The label specified in the R-format must be associated with a remote format.

User Action: Change the label you specified, or create a label and remote format to match the label you specified.

74 The only values that can be converted to pointers are offset variables that have been declared with an area name.

Severity: 3 (Error)

Explanation: 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.

75 A floating-point value is required by this statement, but a noncomputational value has been used. It cannot be converted.

Severity: 3 (Error)

Explanation: The indicated expression or reference has a noncomputational data type that cannot be converted to a floating-point value, yet has been specified in a context that requires a floating-point value.

User Action: Correct the expression so that it specifies a floating-point value or a value that can be converted to floating-point.

76 One of the bounds declared for "*name*" is invalid because the lower bound is greater than the upper bound.

Severity: 3 (Error)

Explanation: 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*.

77 This statement contains an unrecognizable condition name.

Severity: 3 (Error)

Explanation: An ON, SIGNAL, or REVERT statement specifies an invalid condition name.

User Action: Check the statement for a misspelling, and check that the name is a PL/I condition name.

78 A parenthesized name is not valid with this condition.

Severity: 3 (Error)

Explanation: Only the I/O condition names and the VAXCONDITION condition name can specify values.

User Action: Correct the ON condition name in the statement.

79 Excessive complexity of this statement has caused the compiler's parse output stack to overflow. Reduce the complexity and recompile.

Severity: 4 (Fatal)

Explanation: The compiler cannot interpret the statement.

User Action: Rewrite the declaration using two or more statements.

80 "*name*" is an unrecognizable condition prefix name.

Severity: 3 (Error)

Explanation: Unknown condition name encountered in condition prefix.

User Action: Change the condition name.

81 A complex arithmetic value is required by this statement, but a noncomputational value has been used. It cannot be converted.

Severity: 3 (Error)

Explanation: 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.

82 Invalid syntax in this statement. A "name" was found where another token was expected.

Severity: 3 (Error)

Explanation: 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.

83 Invalid syntax in this statement. A name keyword was not found in the expected position.

Severity: 3 (Error)

Explanation: A statement is missing a keyword.

User Action: Correct the syntax of the statement as indicated by the message.

84 Invalid conversion. Noncomputational data cannot be converted to string or arithmetic data.

Severity: 3 (Error)

Explanation: 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.

85 A structure-qualified reference to "name" cannot be resolved to any declaration known to this block.

Severity: 4 (Fatal)

Explanation: 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.

86 A locator-qualified reference to "name" cannot be resolved to any declaration known to this block.

Severity: 4 (Fatal)

Explanation: 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.

87 A structure-qualified reference contains more than 15 qualifying names.

Severity: 3 (Error)

Explanation: 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.

88 A procedure reference contains more than 64 arguments.

Severity: 3 (Error)

Explanation: A CALL statement or a function reference specifies an argument list with more than 64 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 64 arguments.

89 The scale-factor specified in a conversion built-in function exceeds the implementation's limit of -128<=q<=127.

Severity: 3 (Error)

Explanation: 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.

90 The undeclared name "*name*" has been declared as a FIXED BINARY variable in the outermost procedure.

Severity: 2 (Warning)

Explanation: A name that is not declared, and is not a label, has been referenced. The compiler declares it in the outermost procedure with default attributes.

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.

91 "*name*" is a nonbased variable referenced with a locator qualifier.

Severity: 3 (Error)

Explanation: 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.

92 "*name*" is a based variable referenced without a locator qualifier.

Severity: 3 (Error)

Explanation: 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.

93 The base-reference declared for "*name*" contains more * subscripts than this reference has subscripts. This reference cannot be mapped into a valid reference to the base-variable.

Severity: 3 (Error)

Explanation: The number of star (*) subscripts of a variable with the DEFINED storage class does not match the number of star (*) subscripts of the variable on which it is defined.

User Action: Correct the declaration so that it specifies the correct number of subscripts.

94 "*name*" has been passed by-value because its attributes did not match those of the corresponding parameter. Enclose the argument in parentheses to suppress this warning.

Severity: 1 (Informational)

User Action: Check that passing by value is appropriate. Enclose the argument in parentheses to suppress this message.

95 "*name*" has been referenced with too many arguments.

Severity: 3 (Error)

Explanation: 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.

97 The precision arguments of BINARY, DECIMAL, FIXED, FLOAT, DIVIDE, MULTIPLY, ADD, and SUBTRACT built-in functions must be decimal integer constants.

Severity: 3 (Error)

Explanation: 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.

98 "*name*" has been referenced with an argument list which is incompatible with its declaration. An empty argument list is required to satisfy the declaration.

Severity: 3 (Error)

Explanation: 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.

99 This statement contains inconsistent file description attributes or options that conflict with those attributes.

Severity: 3 (Error)

Explanation: 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.

100 "*name*" has been referenced with too few arguments.

Severity: 3 (Error)

Explanation: 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.

101 "*name*" has been referenced with too many empty argument lists.

Severity: 3 (Error)

Explanation: 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.

102 This statement contains duplicate "*name*" options.

Severity: 3 (Error)

Explanation: 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.

103 Invalid option for this statement or invalid syntax in an option. "*name*" is an invalid option.

Severity: 3 (Error)

Explanation: 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.

104 One of the iSUBs used in the base-reference of "*name*" cannot be replaced by a subscript from this reference. This reference cannot be mapped into a valid reference to the base- variable.

Severity: 3 (Error)

User Action: Change the reference to map to the base variable.

105 The precision specified for the result of this built-in function exceeds the implementation's maximum allowed for the resulting data type.

Severity: 3 (Error)

Explanation: An argument list for a built-in function contains an invalid precision argument.

User Action: Correct the precision argument in the built-in function reference that caused the error.

106 A reference to "name" contains a subscript that has not been used as a replacement for an iSUB or an asterisk in the base-reference. This reference cannot be mapped into a valid reference to the base variable.

Severity: 3 (Error)

User Action: Change the reference to map to the base variable.

107 "name" has been referenced with too few subscripts. Asterisk subscripts must be used to reference a cross-section. All subscripted references must have as many subscripts as the array has dimensions - including any inherited dimensions.

Severity: 3 (Error)

Explanation: 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.

108 An array or structure value has been used in a context that requires a scalar value.

Severity: 3 (Error)

Explanation: 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.

109 An assignment target, parameter descriptor, or returns descriptor is not an array or structure of the proper shape to receive the array or structure value being assigned to it.

Severity: 3 (Error)

Explanation: The parameter specification does not correspond to the formal specification.

User Action: Change either the actual parameter or the formal parameter so that they match.

110 A string constant exceeds the implementation's maximum length limit of 4000.

Severity: 3 (Error)

User Action: Shorten the size of the string constant.

111 An operation on two structures is possible only when both structures have the same number of immediate members.

Severity: 3 (Error)

Explanation: The specified operation contains structure operands that do not match.

User Action: Change one or both structures so they match.

112 An operation on two arrays is possible only when both arrays have the same number of dimensions and the exact same lower and upper bounds.

Severity: 3 (Error)

Explanation: The specified operation contains array operands that do not match.

User Action: Change one or both arrays so they match.

113 A bit-string constant contains an invalid digit following the B. Only 1, 2, 3, or 4 are permitted.

Severity: 3 (Error)

Explanation: A bit-string constant is incorrectly specified.

User Action: Determine the correct base for the bit-string constant, and correct the specification.

114 This is an unrecognizable % statement.

Severity: 3 (Error)

Explanation: A statement beginning with a percent sign (%) not a valid PL/I 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.

115 Invalid syntax in %REPLACE statement. The correct syntax is %REPLACE id BY c; where id is an identifier and c is: a character-string constant, a bit-string constant, or an arithmetic constant preceded by an optional minus sign.

Severity: 3 (Error)

Explanation: A %REPLACE statement is specified incorrectly.

User Action: Correct the statement.

116 *name* could not be converted to its internal representation. Check for too many digits, an exponent that is too large or too small, or an invalid character in a string constant.

Severity: 3 (Error)

Explanation: A constant could not be converted to internal representation.

User Action: Check for too many digits, an exponent that is too large or too small, or an invalid character in a string constant.

117 %INCLUDE statements cannot be nested more than 4 levels deep.

Severity: 4 (Fatal)

Explanation: Include files have too many nesting levels.

User Action: Reduce the number of include file nesting levels.

118 Invalid syntax in %INCLUDE statement. The correct syntax is %INCLUDE char-string-constant;

Severity: 3 (Error)

Explanation: A %INCLUDE statement is incorrectly specified.

User Action: Enclose the file name in apostrophes.

119 The first argument to a LBOUND, HBOUND, or DIM built-in function must be an array reference.

Severity: 3 (Error)

Explanation: The argument list for one of the functions listed is incorrectly specified.

User Action: Correct the argument list for the built-in function.

120 Invalid dimension specified in HBOUND, LBOUND, or DIM.

Severity: 3 (Error)

Explanation: 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.

121 The INITIAL attribute specified for "name" contains fewer values than are required to fully initialize the variable.

Severity: 2 (Warning)

Explanation: 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.

122 The ALLOCATION built-in function must reference a nonmember controlled variable.

Severity: 3 (Error)

Explanation: 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.

123 All entries of a multi-entry function must return values whose data types are valid conversion targets for all RETURN statements in the function. This RETURN statement's value cannot be converted to the return type required by one of the entries of this function.

Severity: 3 (Error)

Explanation: This message is issued for a RETURN statement in a procedure with more than one entry point, when the value returned cannot be converted to the type returned by one or more of the entries. It is issued once for each such 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.

124 One of the initial values specified for "*name*" cannot be converted to the type of the variable.

Severity: 3 (Error)

Explanation: 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.

125 More than 64 operands have been used with an operator, function, or call.

Severity: 3 (Error)

Explanation: An expression contains more than 64 operands.

User Action: Simplify the statement in error.

126 Excess initial values have been specified for "*name*".

Severity: 2 (Warning)

Explanation: 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.

127 *name* is an invalid picture.

Severity: 3 (Error)

Explanation: 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.

129 *name* results in a compiled picture that exceeds the implementation's limit. Reduce the size of the picture and recompile.

Severity: 3 (Error)

Explanation: A picture specification is too complex or contains more than 255 characters.

User Action: Correct the picture.

131 *name* contains multiple sign symbols.

Severity: 3 (Error)

Explanation: 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.

132 A static pointer, area or offset, *name*, has been given an initial attribute rather than null or empty. pointers and offsets default to null and areas zeroed control blocks.

Severity: 2 (Warning)

Explanation: A STATIC pointer, offset or area has an INITIAL attribute. Pointers and offsets are initialized to null, and areas to empty, without one.

User Action: Remove the INITIAL attribute, or give it the value NULL or EMPTY.

133 The nesting of DO, PROCEDURE, and BEGIN statements exceeds the implementation's limit of 64.

Severity: 4 (Fatal)

Explanation: 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.

134 One of the targets of this assignment is not a variable or pseudo-variable reference.

Severity: 3 (Error)

Explanation: 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.

135 By-name expansion is invalid because the left-most target of the assignment is a structure that contains more than 20 elementary members.

Severity: 3 (Error)

Explanation: A BY NAME assignment can expand a target structure of at most 20 elementary members.

User Action: Split the assignment, or assign the members separately.

136 A reference to a generic entry name cannot be replaced by any of the alternatives because none match the arguments given in the reference to the generic name.

Severity: 3 (Error)

Explanation: None of the entries in the GENERIC declaration matches the arguments given.

User Action: Check the arguments, or add an entry for them to the GENERIC declaration.

137 A subscript is out-of-range in this statement.

Severity: 2 (Warning)

Explanation: 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.

138 "*name*" is a function that has been called by a CALL statement.

Severity: 3 (Error)

Explanation: 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.

139 "name" is a subroutine that has been called as a function.

Severity: 3 (Error)

Explanation: The statement contains a reference to a procedure that does not have the RETURNS attribute.

User Action: If the invoked procedure is a subroutine, correct the statement in error so that the procedure is invoked with a CALL statement. Otherwise, delete the RETURNS attribute from the PROCEDURE statement of the procedure so that it can be invoked by a function reference.

140 A dope-vector exceeds the implementation's limit of 512 bytes. Either eliminate the use of * extents in this entry point, or pass structures by passing pointers to them.

Severity: 4 (Fatal)

Explanation: A structure parameter has so many members with asterisk extents that its descriptor cannot be represented.

User Action: Simplify the parameters.

141 "name" is an undeclared parameter. It has been declared in its containing block and will acquire default attributes.

Severity: 1 (Informational)

Explanation: A name in a parameter list is not declared with a data type, so the compiler gives it default attributes.

User Action: Declare the parameter with the appropriate data type attributes.

143 "name" has been declared without a data type. A default type of FIXED BIN(15) has been supplied.

Severity: 2 (Warning)

Explanation: The name is declared without a data type attribute, so the compiler gives it a default data type.

User Action: If the default data type is what you want, no action is needed. Otherwise, give the declaration a data type.

145 "name" has been declared with an INITIAL attribute. An INITIAL attribute cannot be specified for variables of this storage class.

Severity: 3 (Error)

Explanation: The INITIAL attribute was given for a DEFINED variable or 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.

147 "*name*" is an entry or file constant and as such it cannot be declared with a DIMENSION attribute.

Severity: 3 (Error)

Explanation: 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.

148 An expected optionally signed integer was not found.

Severity: 3 (Error)

Explanation: A nonconstant expression is specified in a context that requires an integer constant.

User Action: Specify an integer constant.

150 "*name*" has been declared FLOAT with a scale-factor. The scale-factor will be ignored.

Severity: 3 (Error)

Explanation: 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.

151 "*name*" has been declared with a negative size. A size of one has been supplied.

Severity: 3 (Error)

Explanation: 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.

152 "*name*" is undeclared and has been used in an ALLOCATE or LOCATE statement as the name of a based or controlled variable.

Severity: 3 (Error)

Explanation: 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.

153 "*name*" cannot be allocated because no SET option or implied SET option is available.

Severity: 3 (Error)

Explanation: 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.

154 "*name*" cannot be allocated because no IN option or implied IN option is available.

Severity: 3 (Error)

Explanation: The variable is BASED on an offset, or allocated in an area, but the ALLOCATE statement has no IN option and the declaration does not name an area.

User Action: Add an IN option to the ALLOCATE statement, or name the area in the declaration of the offset.

155 "*name*" cannot be allocated because the SET option or implied SET option does not reference a pointer or offset variable.

Severity: 3 (Error)

User Action: Correct the SET option, or the declaration of the BASED variable, so that it names a pointer or offset variable.

156 "*name*" cannot be allocated because the IN option does not specify an area value.

Severity: 3 (Error)

Explanation: 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.

157 "*name*" cannot be allocated because it is neither a based nor a controlled variable. Only based variables can be allocated by a LOCATE statement.

Severity: 3 (Error)

Explanation: An ALLOCATE or LOCATE statement names a variable that is neither BASED nor CONTROLLED.

User Action: Correct the statement, or the declaration of the variable.

158 All RETURN statements in a function must return values.

Severity: 3 (Error)

Explanation: 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.

159 A RETURN statement in a subroutine cannot return a value.

Severity: 3 (Error)

Explanation: 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.

160 A RETURN statement cannot be contained within an on-unit.

Severity: 3 (Error)

Explanation: 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.

161 A FREE statement must free a nonmember BASED or CONTROLLED variable.

Severity: 3 (Error)

Explanation: 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.

162 In order to prevent the creation of an array whose elements are of unequal length, a default length of 128 has been used for the element length of an array temporary.

Severity: 2 (Warning)

User Action: None.

163 A RETURNS option or attribute must specify a scalar, array, or structure. It cannot specify a list of values. Check for missing or incorrect level-numbers.

Severity: 3 (Error)

Explanation: Invalid return value detected.

User Action: Change the return value to specify a scalar, array, or structure.

164 Invalid conversion required by an assignment, return, or argument-list.

Severity: 3 (Error)

Explanation: The indicated value specified in an assignment, a RETURN statement, or in an argument list does not have a data type that is valid for conversion to the data type of the other operand, the return descriptor, or the called routine.

User Action: Determine the appropriate data type, and correct either the indicated value or the operand, descriptor, or routine declaration.

165 An offset has been declared relative to a nonarea variable.

Severity: 3 (Error)

Explanation: An offset is declared with the OFFSET attribute naming a variable that is not an area.

User Action: Name an area in the OFFSET attribute.

166 Subscripts are not valid in the reference of a REFER option. The referenced variable must not be an array or member of a dimensioned structure.

Severity: 3 (Error)

Explanation: The REFER option can be used only with non-dimensioned structures declared with the BASED attribute.

User Action: Change the referenced variable, or remove the REFER option.

167 "name" is not a built-in function known to this implementation. If this is an external entry, it must be declared by a DECLARE statement with an ENTRY attribute.

Severity: 3 (Error)

Explanation: 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.

168 The argument of ADDR must be a reference to a variable.

Severity: 3 (Error)

Explanation: 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.

169 A built-in function that produces a floating-point result cannot specify a scale-factor.

Severity: 3 (Error)

Explanation: 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.

170 Offset/pointer conversion is possible only if the offset has an associated area value.

Severity: 3 (Error)

Explanation: 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.

171 The argument of UNSPEC must be a reference to a scalar variable or a reference to an element of an array of scalar values.

Severity: 3 (Error)

Explanation: 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.

172 The argument of ADDR cannot be an unaligned bit-string or an aggregate of unaligned bit-strings.

Severity: 1 (Informational)

Explanation: 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.

173 Possible compiler error. PASS2 attempted to free an already freed value node. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 3 (Error)

Explanation: The compiler detected an internal error.

User Action: Fix any other errors generated during the compilation, and recompile. If this message persists, report the problem to Endless Software Solutions.

174 The operands of ^<, ^>, >, <, >=, or <= must be real arithmetic or string values.

Severity: 3 (Error)

Explanation: A noncomputational value, such as a pointer or a file, is an operand of a relational operator other than = or ^=.

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.

175 Areas cannot be compared in any context. Arrays or structures cannot be compared in IF, WHILE, or UNTIL contexts because the result of such a compare is not a scalar value.

Severity: 3 (Error)

Explanation: 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.

176 The name given in a CONDITION() clause must be the name of a programmer-defined condition. Such names can be declared with the condition attribute.

Severity: 3 (Error)

Explanation: 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.

177 The name given in an I/O condition must be the name of a file value.

Severity: 3 (Error)

Explanation: 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.

178 An unconnected array cannot be referenced in an ADDR, INTO, FROM, STRING pseudo-variable, base reference of a DEFINED variable, or argument that is to be passed by-reference to a parameter that has constant extents. In the latter case, the parameter must have asterisk extents.

Severity: 3 (Error)

Explanation: 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.

179 The undeclared name "*name*" has been contextually declared in this block. It will acquire default attributes.

Severity: 1 (Informational)

User Action: Check that the default attributes give the right results. You may want to declare the name to avoid the message.

180 The undeclared name "*name*" has been implicitly declared in the outermost block. It will acquire default attributes.

Severity: 1 (Informational)

User Action: Check that the default attributes give the right results. You may want to declare the name to avoid the message.

181 The BY NAME option is invalid on this statement because the structures referenced herein do not have at least one member whose qualified name is the same for all structures.

Severity: 3 (Error)

User Action: Correct the structures, or use separate assignments instead of BY NAME.

182 A reference in a GET DATA or PUT DATA must identify a variable.

Severity: 3 (Error)

Explanation: The GET DATA or PUT DATA statement does not reference a variable.

User Action: Correct the statement.

183 Subscripts are not valid in a GET DATA list or in a LIKE attribute.

Severity: 3 (Error)

User Action: Remove the subscripts.

184 "*name*" has one or more structure members as well as a LIKE attribute. A LIKE attribute cannot be used if the structure has members of its own.

Severity: 3 (Error)

Explanation: 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.

185 The LIKE attribute specified for "*name*" cannot be resolved to a declaration known within this block.

Severity: 3 (Error)

Explanation: 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.

186 The LIKE attribute specified for "*name*" references a structure or member of a structure that was produced by the expansion of another LIKE attribute.

Severity: 3 (Error)

Explanation: The LIKE attribute can reference major or minor structures known to the current block, but cannot directly or indirectly reference a structure resulting from the expansion of another LIKE attribute.

User Action: You may need to specifically declare the structure.

187 The LIKE attribute specified for "*name*" does not reference a structure.

Severity: 3 (Error)

Explanation: The LIKE attribute must reference a major and minor structure 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.

188 "*name*" has been given the ERROR attribute by a DEFAULT statement.

Severity: 3 (Error)

Explanation: A DEFAULT statement with the ERROR attribute applies to the name, which means that it must be declared explicitly.

User Action: Declare the name with explicit attributes.

189 The declaration of "*name*" contains attributes that conflict with those of an applicable DEFAULT statement. To avoid this message, make the predicate of the DEFAULT state- ment more explicit. The DEFAULT statement has been ignored.

Severity: 1 (Informational)

User Action: To avoid this message, make the predicate of the DEFAULT statement more explicit. The DEFAULT statement has been ignored.

190 DEFAULT statements must appear in an external procedure, They cannot be contained in internal procedures or begin-blocks.

Severity: 3 (Error)

User Action: Move the DEFAULT statement to an external procedure.

191 Excess initial values have been specified for "*name*".

Severity: 3 (Error)

Explanation: 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.

192 Incomplete attributes have been specified for "*name*".

Severity: 3 (Error)

Explanation: Something is missing in a declaration.

User Action: Correct the declaration.

193 "*name*" has been declared with REFER options but is not a based variable.

Severity: 3 (Error)

Explanation: 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.

194 An array bound, string length, structure member offset, array element offset, or the size of a variable exceeds 231 or was specified with an invalid constant.**

Severity: 3 (Error)

Explanation: 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.

195 "*name*" has been declared with a length greater than 32767. A length of one has been supplied.

Severity: 3 (Error)

Explanation: 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.

196 "*name*" has been declared with a variable iteration factor. An iteration factor of one has been supplied.

Severity: 3 (Error)

Explanation: 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.

197 "*name*" has been declared with a nonconstant initial value. Static variables must have constant initial values.

Severity: 3 (Error)

Explanation: A static variable is incorrectly initialized.

User Action: Correct the declaration so that it uses only constant values in the INITIAL attribute.

198 "*name*" has been declared with an INITIAL attribute containing more than 32 levels of nesting.

Severity: 3 (Error)

User Action: Simplify the INITIAL attribute.

199 The initial value specified for "*name*" exceeds the implementation's limit of 4096 bytes.

Severity: 3 (Error)

User Action: Use a smaller initial value, or assign the value in executable statements.

200 Possible compiler error. This phase has *name* unclaimed nodes. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 2 (Warning)

Explanation: The compiler detected an internal error.

User Action: Correct all other errors reported during the compilation and recompile. If this message persists, report the problem to Endless Software Solutions.

201 The entry name selected by a reference to "*name*" has not been declared as an ENTRY. All entry names in a GENERIC attribute must be declared ENTRY either by a DECLARE or PROCEDURE or ENTRY statement.

Severity: 3 (Error)

Explanation: You referenced an undeclared entry name in a GENERIC attribute.

User Action: Declare the entry name.

202 A value used in this statement cannot be converted to the data type required by the context in which it is used.

Severity: 3 (Error)

Explanation: 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.

213 String constants used with repetition factors in an INITIAL attribute must be enclosed in parentheses in order to be compatible with standard PL/I.

Severity: 2 (Warning)

User Action: Enclose the string constant in parentheses, as in `INITIAL ((10) ('A'))`.

214 "*name*" has been truncated to eight characters. Do not use external names longer than eight characters.

Severity: 1 (Informational)

User Action: Use a shorter external name.

221 Excessive size and complexity of a procedure produced more than 32767 operators and exceeded the compiler's limit for a single procedure. Reduce the size or complexity of the procedure that contains this statement.

Severity: 4 (Fatal)

Explanation: This statement pushes the procedure over the compiler's limit of 32767 constants, identifiers, and punctuation symbols, including operators.

User Action: Simplify the procedure.

222 '*name*' must have an INITIAL((n)EMPTY) attribute, where n is the number of elements in the array.

Severity: 3 (Error)

User Action: Add the INITIAL attribute to the declaration.

224 A member of a dimensioned structure is considered to be an interleaved array and as such cannot be used in an aggregate assignment or other context that requires connected storage.

Severity: 3 (Error)

User Action: Change the code to specify another operand to the aggregate assignment, or change the declaration of the specified array.

225 Possible compiler error. The code generator has found an invalid data type. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 3 (Error)

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

226 Possible compiler error. The code generator has overflowed its stack. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 4 (Fatal)

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

230 Possible compiler error. The code generator tried to free an already freed value node. Correct all program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 3 (Error)

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

234 A complex variable cannot be used as an index of a DO statement.

Severity: 3 (Error)

User Action: Use another variable as the index for the DO statement.

238 A Built-in function or concatenation produced a result string whose length exceeds 32767 or is negative.

Severity: 3 (Error)

User Action: Supply shorter values to the built-in function.

244 The name built-in function requires a variable reference as its argument.

Severity: 3 (Error)

Explanation: The argument of the built-in function, for example SIZE, is not a reference to a variable: it might be a constant or an expression.

User Action: Supply a valid argument.

246 "name" has a data type that is not supported in this implementation.

Severity: 2 (Warning)

User Action: Use another data type.

247 The denominator in a divide operation has been evaluated as a constant of zero value.

Severity: 3 (Error)

Explanation: A division has a divisor that the compiler has evaluated as the constant zero.

User Action: Correct the expression.

248 A descriptor for the entry, "*name*" referenced on this line has been declared with a VALUE attribute. This is obsolete and not portable to other implementations. Use options(c) on the declaration.

Severity: 1 (Informational)

Explanation: An entry has a parameter descriptor with the VALUE attribute. This form is obsolete.

User Action: Declare the entry with OPTIONS(C) instead.

249 A descriptor for the entry, "*name*" has a data type not suitable for for the VALUE attribute, it requires a descriptor. Remove the VALUE attribute from the declaration.

Severity: 3 (Error)

Explanation: The VALUE attribute is given for a parameter whose data type has to be passed with a descriptor.

User Action: Remove the VALUE attribute from the declaration.

358 A substring reference to "*name*" specifies an invalid offset or length value. The two offset expressions must satisfy the following: 1 .LE. offset1 .LE. offset2 .LE. length-of-item

Severity: 3 (Error)

User Action: Change the offset or length of one of the two expressions.

417 A precision/length exceeds the implementation limit of 32767.

Severity: 3 (Error)

User Action: Change the specified precision or length so it no longer exceeds 32767.

430 The subroutine entry "*name*" is contained in a function procedure. This does not conform to the ANSI definition.

Severity: 1 (Informational)

User Action: If you plan on porting this application, check that your target platform supports the specified statement.

431 The function entry "*name*" is contained in a subroutine procedure. This does not conform to the ANSI definition.

Severity: 1 (Informational)

User Action: If you plan on porting this application, check that your target platform supports the specified statement.

432 The structure variable "*name*" is being compared for equality to the null pointer but does not have a locator qualifier in its declaration.

Severity: 3 (Error)

Explanation: Only structure variables containing locator qualifiers in their declarations can be compared to the null pointer.

User Action: Add a locator qualifier to the declaration or change the comparison.

435 A reference to a cross section of an array of structures is not permitted in IBM compatibility mode. Remove the asterisks and surrounding parentheses from the reference to *name*.

Severity: 1 (Informational)

User Action: Remove the asterisks and the parentheses around them, or compile with another dialect.

436 This %replace statement is a duplicate of one that appeared previously. Please remove one of them.

Severity: 1 (Informational)

User Action: Remove one of the duplicate %replace statements.

437 Invalid syntax in a %replace statement. Check to see if the identifier appeared in another %replace statement with a different replace value.

Severity: 3 (Error)

Explanation: You cannot specify the same identifier in two %replace statements that give different values.

User Action: Check to see if the identifier appeared in another %replace statement with a different replace value, and correct the code.

439 The implementation-defined limit of 99 include files has been exceeded.

Severity: 4 (Fatal)

Explanation: You cannot include more than 99 files.

User Action: Nest or combine some of the files, if possible, to somehow reduce the number.

444 The token, "*name*" is not a valid environment option.

Severity: 3 (Error)

Explanation: The specified token is not a valid environment option.

User Action: Check the reference manual for valid environment options, and correct the expression.

445 "*name*" which has been declared with the LIKE or TYPE attribute must have a level number. The default value is set to level 1.

Severity: 2 (Warning)

User Action: Check that level 1 is right. You may want to give the level number to avoid the message.

446 A value in this expression is being converted to type POINTER. This is an extension to the ANSI standard.

Severity: 1 (Informational)

User Action: None. If the program is to be ported, check that the other compiler allows the conversion.

447 The only arithmetic permitted on pointers is the addition or subtraction of a fixed binary expression.

Severity: 4 (Fatal)

Explanation: The code attempts to do an arithmetic operation on a pointer other than addition or subtraction.

User Action: Remove the specified operation.

448 A pointer-valued expression is being assigned to a fixed binary(15) reference, "name", which will result in loss of significance.

Severity: 1 (Informational)

User Action: Check that the assignment does what you intend.

449 "name" has been declared with the CONTROLLED attribute and can not be allocated in an area or with a set option. Change the the declaration from CONTROLLED to BASED.

Severity: 2 (Warning)

Explanation: 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.

450 Invalid syntax in this statement. An unexpected token has been encountered.

Severity: 3 (Error)

Explanation: The compiler encountered a token it did not expect.

User Action: Remove or change the token, or otherwise fix the code so that the token is appropriate.

452 "name" has been declared as a register. Only integer values, in the range appropriate to this machine are permitted in the register specification. Consult the reference manual for the permissible values.

Severity: 3 (Error)

User Action: Specify an integer of the correct value in the register specification.

453 Only fixed binary and pointer data types may be assigned to a register.

Severity: 3 (Error)

Explanation: The code attempts to assign something other than a fixed binary or pointer to a register.

User Action: Change the code so that you only assign appropriate types of data to the register.

455 The entry declaration is missing a parameter descriptor, which will suppress type checking in ibm compatibility mode, but will default to fixed bin(15) in ANSI mode.

Severity: 1 (Informational)

User Action: Check that the default gives the right results. You may want to add the parameter descriptor to avoid the message.

456 "*name*" has been declared with an incorrect precision. The precision must be an integer greater than zero. "*name*" will be given a default precision as described in the reference manual.

Severity: 1 (Informational)

User Action: Check that the default precision gives the right results, and give a valid precision.

457 An entry appears in the generic descriptor list of "*name*". Entries used in this manner must not have a descriptor list of their own. Remove the descriptor list.

Severity: 2 (Warning)

User Action: Remove the descriptor list.

458 In the declaration of the generic intrinsic procedure "*name*", one of the members of the generic descriptor list has an invalid attribute. Consult the table of permissible attributes in the reference manual.

Severity: 2 (Warning)

User Action: Correct the invalid attribute.

459 If the level number of a structure is specified in an ALLOCATE statement, it must be level 1.

Severity: 3 (Error)

Explanation: The compiler detected a level number other than 1 in an ALLOCATE statement.

User Action: Correct the code to specify level 1, after checking that the structure is correct.

460 A controlled variable in this ALLOCATE statement has previously been declared without the CONTROLLED attribute in the procedure "*name*". The previously declare variable will acquire the CONTROLLED attribute.

Severity: 2 (Warning)

User Action: Check that the CONTROLLED attribute gives the right results. You may want to declare the variable CONTROLLED to avoid the message.

461 The controlled variable, "*name*", has not been declared. It will acquire the attributes from the ALLOCATE statement in which it appears.

Severity: 1 (Informational)

User Action: Check that the attributes give the right results. You may want to declare the variable to avoid the message.

462 The only attributes permitted in the ALLOCATE controlled variable are: dimension, CHAR([length]*), BIT([length]*), G([length]*), AREA([size]*), INIT(i[,j]...) or INIT entry[(arg1[,arg2]...)]. If * is given, then value is taken from current generation. If none exists, the program is in error.

Severity: 3 (Error)

User Action: Use only the attributes listed.

463 The variable, "*name*", has a * extent and is not a parameter. It will acquire the CONTROLLED attribute.

Severity: 1 (Informational)

User Action: Check that the CONTROLLED attribute gives the right results. You may want to declare the variable CONTROLLED to avoid the message.

464 The variable, "*name*", has a * extent, is not a parameter and has no data type. It will acquire CONTROLLED and FIXED BIN(15) attributes.

Severity: 2 (Warning)

User Action: Declare the variable with a data type, and as CONTROLLED if that is what you intend.

465 The data type given for the controlled variable, "*name*", in its declaration does not agree with the attributes given in the ALLOCATE statement.

Severity: 3 (Error)

User Action: Change either the data type of the controlled variable or the attributes of the ALLOCATE statement.

466 A call to the function, "*name*", which takes no arguments does not use the preferred syntax *name*(). The meaning of the context in which it is used would be much clearer in the latter form.

Severity: 1 (Informational)

User Action: Use the specified syntax to avoid getting this message in the future.

467 The label array "*name*" is an automatic array of label variables, elements of which are use as statment prefixes. If the label array is never modified, remove the declaration. Otherwise, this will result in terribly inefficient code.

Severity: 1 (Informational)

User Action: Remove the declaration if the label array is never modified. Otherwise, no action needed.

468 "*name*" has been declared without a data type but with an initial attribute. It will acquire the type of the initial value.

Severity: 1 (Informational)

User Action: Check that the data type is right. You may want to give the declaration a data type to avoid the message.

471 Invalid syntax in a %DCL statement. An identifier was expected after the comma.

Severity: 3 (Error)

User Action: Correct the statement.

472 "*name*" is being activated, but no replacement has yet been given.

Severity: 1 (Informational)

Explanation: A %ACTIVATE statement names a preprocessor identifier that has no value yet.

User Action: Give the identifier a value before activating it, or check the order of the preprocessor statements.

473 "*name*" is being declare without and attribute in a %DCL statement. Please specify one of the following: CHAR, FIXED, ENTRY or BUILTIN (case unimportant).

Severity: 3 (Error)

User Action: Specify one of the listed attributes in your %DCL statement.

474 Syntax error in a DEFAULT statement in IBM compatibility mode, missing left parenthesis in a VALUE specification.

Severity: 3 (Error)

User Action: Add the left parenthesis.

477 Possible compiler error. Unexpected condition signaled. Correct any program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 4 (Fatal)

Explanation: A condition was raised inside the compiler that it did not expect.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

478 Internal compiler error detected. Unexpected operator encountered during code generation.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

479 Compilation terminated by user.

Severity: 4 (Fatal)

Explanation: The compilation was interrupted, for example from the keyboard.

User Action: None.

481 Cannot open include file "*name*".

Severity: 3 (Error)

Explanation: The file named in a %INCLUDE statement could not be opened.

User Action: Check the name of the file, the directories searched for include files (the -I option), and the permissions of the file.

482 Fatal internal compiler error detected. Correct any program errors and recompile. If this message persists, contact compiler maintenance personnel.

Severity: 4 (Fatal)

Explanation: An internal error occurred in the compiler.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

483 Internal compiler error detected. Temp value stack overflow.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

484 Internal compiler error detected. Temp value stack underflow.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

485 Internal compiler error detected. Virtual registers exhausted.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

488 Internal compiler error detected. Unexpected condition signaled in codegen.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

489 Internal compiler error detected. Unexpected condition signaled in codegen-select_case.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

490 Internal compiler error detected. Unexpected condition signaled.

Severity: 2 (Warning)

Explanation: An internal error occurred in the compiler.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

491 The value of the AUTOMATIC or DEFINED variable "*name*" is used in an extent expression or initial-value expression in the declaration of an AUTOMATIC or DEFINED variable in the same block.

Severity: 2 (Warning)

Explanation: The declaration of an AUTOMATIC or DEFINED variable uses, in an extent or initial value, the value of another AUTOMATIC or DEFINED variable of the same block. The order in which the variables are allocated and initialized is not defined, so the value may not be set yet.

User Action: Declare the variable used in the expression in an enclosing block, or assign the value in executable statements.

494 Internal compiler error detected. Symbol *name* is already linked.

Severity: 2 (Warning)

Explanation: An internal error occurred in the compiler.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

496 Use of parentheses surrounding the like reference is not standard and will render the program less portable.

Severity: 1 (Informational)

User Action: Remove the parentheses.

497 This is an invalid EXEC statement. Only SQL, CICS and DL1 are supported.

Severity: 3 (Error)

Explanation: An EXEC statement names something other than SQL, CICS or DL1.

User Action: Correct the EXEC statement.

499 Internal compiler error detected. SP reset point not allocated.

Severity: 2 (Warning)

Explanation: An internal error occurred in the code generator.

User Action: Correct any other errors in the program and recompile. If the message persists, report the problem to Endless Software Solutions.

502 The based structure member, *name* has been used in an extent expression in an array or string member of the same structure. This is invalid PL/I and should be replaced with refer extents. Such structures may not be allocated. The name, *name*, must have a unique spelling within this scope.

Severity: 1 (Informational)

User Action: Use REFER options for the extents of the members.

503 Use of the built-in function *name* requires the Digital dialect option.

Severity: 4 (Fatal)

Explanation: The built-in function exists only in the DEC dialect.

User Action: Compile with -D dec, or do not use the built-in function.

504 Use of the pseudo-variable *name* requires the Digital dialect option.

Severity: 4 (Fatal)

Explanation: The pseudovvariable exists only in the DEC dialect.

User Action: Compile with -D dec, or do not use the pseudovvariable.

507 The external procedure name "main" will conflict with the library function "main" supplied in the UniPrise PL/I run-time library, inhibiting initialization of the run-time.

Severity: 1 (Informational)

Explanation: An external procedure is named `main`. The run-time library has its own function of that name, which starts the run-time environment.

User Action: Rename the procedure.

508 The UniPrise PL/I symbolic debugger does not support programs containing executable statements in included text files.

Severity: 1 (Informational)

Explanation: The source program has executable statements in a file included with `%INCLUDE`. The debugger cannot show them.

User Action: To debug those statements, move them into the main source file.

510 The condition "*name*" is always enabled and cannot be disabled by a condition prefix.

Severity: 3 (Error)

User Action: Remove the condition from the prefix.

511 The condition "*name*" is not allowed as a prefix.

Severity: 3 (Error)

User Action: Remove the condition from the prefix.

512 Use of the *name* requires the Digital dialect option.

Severity: 4 (Fatal)

Explanation: The feature exists only in the DEC dialect.

User Action: Compile with `-D dec`, or do not use the feature.

513 Use of the *name* requires the IBM dialect option.

Severity: 4 (Fatal)

Explanation: The feature exists only in the IBM dialect.

User Action: Compile with `-D ibm`, or do not use the feature.

514 The *name* is not supported in this implementation.

Severity: 3 (Error)

User Action: Do not use the feature.

515 Invalid *-extent in a RETURNS attribute for *name*.

Severity: 3 (Error)

Explanation: The RETURNS attribute gives an asterisk extent that is not allowed.

User Action: Give a constant extent.

516 The TYPE attribute specified for "*name*" cannot be resolved to a declaration known within this block.

Severity: 3 (Error)

Explanation: The name in the TYPE attribute is not declared in, or known to, the block.

User Action: Check the name, and declare the type it refers to.

517 The TYPE attribute specified for "*name*" is a circular definition.

Severity: 3 (Error)

Explanation: The TYPE attribute refers, directly or through other TYPE attributes, back to the name being declared.

User Action: Correct the declarations.

Index

Symbols

\$signaldef, 82
%PROCEDURE statement, 9
%RETURNS statement, 10

A

Access modes, 26
 keyed, 26
 sequential, 26
 using record identification, 27
Access privileges, 53
Activation records of ON-units, 85
ADDR built-in function, 97
ANY attribute, 23, 95, 98
 using with VALUE, 98
ANY OPTIONAL, 99
ANYCONDITION ON-unit
 affect of nonlocal GOTO, 89
 calling during unwind, 89
 located in search for ON-units, 86, 87
 STOP statement in, 89
APPEND ENVIRONMENT option, 41
 example, 29
Archive libraries, 7
AREA condition, 92
Arguments
 optional, 99
 passing
 by reference, 95
 by value, 97
 specifying pointer values, 96
 truncating lists of, 99
Arrays
 passing as arguments, 95
 by reference, 95
 to FORTRAN, 97
Ascending sort, 66
Assembly language, 3

Attributes for file access, 26

B

Binary, 3
Bit strings
 passing by reference, 95
 passing by value, 97
Boolean expressions
 in ENVIRONMENT options, 36
 in I/O statement options, 55
Buffers, file, 60
Built-in functions
 ADDR, 97
 condition-handling, 91
 ONCODE, 91
 ONFILE, 93
 ONKEY, 93
 preprocessor, 11
Built-in subroutines
 file-handling, 60
 FLUSH, 60
 RESIGNAL, 37, 76, 78
 REWIND, 60

C

C, 4
Call frame
 for ON-unit, 85
 removing from call stack, 80
Call stack
 execution of ON-unit, 85
 searching for ON-units, 33, 85, 87, 87
 during unwind, 80
 effect of RESIGNAL, 28, 79
 unwind, 80, 80
CALL statement
 external routines, 100
 FORTRAN example, 97
 non-PL/I routines, 94
CANCEL_CONTROL_O option, 55
Character strings
 as arguments to ENVIRONMENT options, 36
 passing as arguments, 95

- reading and writing
 - fixed-length, 28
 - varying-length, 28
- chmod command, 54
- Cleanup procedures, 90
- CLOSE statement, 18
 - specifying ENVIRONMENT options, 35
- Compiler
 - causes of warnings, 15
 - components of, 2
 - diagnostic message format, 14
 - diagnostic messages, 129
 - driver options, 4
 - drivers for other languages, 4
 - error severity, 14
 - function of, 2
 - processing phases, 3
 - syntax error detection, 15
- CONDITION condition, 92
- Condition handlers, 81
 - compared to ON-units, 81
 - courses of action, 78
 - default, 86
 - defaults in routines, 87
 - effects of using, 78
 - multiple active, 88
 - resignaling a condition, 37, 76, 78, 79
 - search order, 33, 85
 - signal(3), 81
 - unwinding the call stack, 80
 - using built-in functions, 91
 - using STOP statement in, 81
 - VAXCONDITION, 76
- Condition Handling Standard support, 94
- Condition values
 - file errors, 23
 - user-defined, 76
- CONSECUTIVE ENVIRONMENT option, 42
- Constants, file, 18

D

- Default condition handling
 - main procedure, 86

- non-main procedure, 87
- DEFAULT_FILE_NAME ENVIRONMENT option, 43
- DELETE ENVIRONMENT option, 43
- Descending sort, 66
- DIRECT file access, 26
- Driver options, 4
- Dummy arguments
 - for by-value arguments, 98
 - passing by reference, 96
- Duplicate keys, errors in, 23

E

- ENDFILE condition, 22, 92, 93
- ENDPAGE condition, 22, 92
 - action by default handler, 86
- ENTRY attribute, declaring non-PL/I routines, 97
- Entry name, as procedure argument, 98
- ENVIRONMENT options, 35
 - APPEND, 41
 - CONSECUTIVE, 42
 - DEFAULT_FILE_NAME, 43
 - DELETE, 43
 - F, FB, FS, and FSB, 42, 44
 - FIXED_LENGTH_RECORDS, 42, 44, 45, 52
 - GROUP_PROTECTION, 45, 81
 - MAXIMUM_RECORD_SIZE, 46, 48
 - OWNER_PROTECTION, 47
 - RECORD_ID_ACCESS, 48
 - SCALARVARYING, 49
 - SHARED_READ, 50
 - specifying, 35
 - specifying arguments, 36, 36
 - summary, 37, 76
 - SUPERSEDE, 50
 - TEMPORARY, 51
 - V, VB, VS, and VSB, 52
 - WORLD_PROTECTION, 52
- Environment variables, 20
- EQUALS (sort option), 66
- ERROR condition, 92
 - action by default handler, 86
 - default ON-unit action, 78
 - for file errors, 22

- signaled
 - by default ON-unit, 86
- ERROR ON-unit, 90
- Errors
 - compiler messages, 129
 - ENVIRONMENT options, 36, 76
 - file
 - default handling, 24
 - error handler, 23
 - handling, 36, 76
 - file, 22
 - of file-related error, 93
 - ON conditions, 36, 76
 - message format in compiler, 14
 - relative file, 33
 - run-time, 15
 - run-time messages, 108
 - severity, 14
 - syntax, 15
 - when passing arguments, 96
- Executable binary, 3
- Executing a PL/I program, 15
- Exit status
 - run-time panic, 127
 - unhandled condition, 108
- External procedures as arguments, 98
- External routines
 - calling, 100
 - non-PL/I, 94

F

- F, FB, FS, and FSB ENVIRONMENT options, 42, 44
- FAST_DELETE option, 55
- Fatal severity condition handling, 86
- FILE attribute, 18
- File constants, 19
- File specifications
 - completing, 20
 - expanded, 21
 - for error, 93
 - invalid, 20
 - TITLE option, 19
- Files, 18

- access modes and PL/I attributes, 26
- access privileges, 53
- accessing, 18
- built-in subroutines for, 60
- closing, 18
- creating, 29
- declaring, 18
- default version numbers, 21
- error conditions in, 22
- handling errors, 23
- keyed, error handler ex., 23
- opening
 - ENVIRONMENT options, 36
 - positioning, 18
- organization, 25
- ownership, 53
- prefix for sort files, 68
- process permanent, 20
- protection, 53
 - specifying, 53
- reading and writing, 32, 32, 58
- record, 25
- relative, 30, 33
- signaling ONKEY built-in function, 93
- stream, 29
- FILSZ (sort option), 66
- FINISH condition
 - signaled
 - by default handler, 86
 - by STOP statement, 81
 - STOP statement in ON-unit, 89
- Fixed-length records, 28
- FIXEDOVERFLOW condition, 92
- FIXED_LENGTH_RECORDS ENVIRONMENT option, 28, 42, 44, 45, 52
- FLUSH built-in subroutine, 60
- Formats, record, 27
- Fortran, 4
- FORTTRAN routines, passing arrays to, 97
- Functions, calling, 100

G

- GET statement

- default file title, 21
 - NO_ECHO option, 56
 - suppressing display of input, 56
 - with PROMPT option, 57
- Global symbols
 - condition value, 92
 - ONCODE values, 92
 - reference in ON-unit
 - example, 34
 - referencing in ON-unit, 23, 92
- GOTO statement, nonlocal, 80
- GROUP_PROTECTION ENVIRONMENT option, 45, 53, 81

I

- I/O
 - overview of operating system features, statement options, 56, 58
- I/O Statement options
 - format, 55
- I/O statement options, 55, 56
- IBM dialect, 22
 - File environment options, 28, 28
- Include files
 - input parameter declarations, 16
- Indexed files
 - error messages, 127
- Input files, defining, 19
- Input/Output
 - file specifications, 19
 - record file, 25
 - stream file, 29
 - to a file, 18

K

- KEY condition, 22, 93
 - sample ON-unit, 23, 34
- KEY option, in relative file, 32
- KEYED attribute
 - creating a relative file with, 31
 - example, 31
 - file access, 26
- Keys

- file access using, 26
 - handling errors in, 24, 34, 87

L

- Labels
 - preprocessor, 11
- Length of variable length records, 28
- Link edit, 3
- Linking objects, 6
 - specifying run-time library, 7

M

- Main procedure condition handling, 86
- MAXIMUM_RECORD_SIZE ENVIRONMENT option, 28, 28, 31, 46, 48
- Message level (for sorting), 69
- Messages
 - compiler, 129
 - run-time, 108
- Multiple active conditions, 88

N

- NAME condition, 22
- NOEQUALS (sort option), 66
- Non-PL/I routines, 95
 - calling, 94, 100
- Nonlocal GOTO, 80
- NO_ECHO option, 56
- Null statements, in ON-units, 91

O

- Object file, 3
- Object libraries, 7
- ON conditions, 22, 36, 76
- ON-units
 - call frame created for, 85
 - compared to condition handlers, 81
 - courses of action, 78
 - examples of, 90
 - for KEY condition
 - in indexed file, 23
 - in relative file, 34
 - referencing global symbols, 91

- examples, 23, 34
 - scope of, 90
 - searching of call stack, 33, 85
- ONCODE built-in function, 23, 92, 95
 - return value global symbol names, 92
 - STATUS variable, 90
 - testing, 79
 - values in KEY ON-unit, 24, 34
- ONFILE built-in function, 23, 93
- ONKEY built-in function, 23, 93
 - ONCODE values, 24, 34
- OPEN statement, 18
 - specifying TITLE option, 19
- OPTIONAL attribute, 99
- Options
 - ENVIRONMENT, 35
 - summary, 37, 76
- OPTIONS MAIN default condition handling, 86
- OPTIONS option, 55
- OPTIONS(VARIABLE) option, 99
- OUTPUT attribute
 - creating a new file, 29
 - example, 31
- Output, record, 25
- OVERFLOW condition, 92
- OWNER_PROTECTION ENVIRONMENT option, 47, 53

P

- Parameters
 - attributes, 98
 - passing to non-PL/I routines, 95
 - VALUE attribute, 97
- Pascal, 4
- pl1 command, 2, 2
 - error messages, 14
 - linking objects, 6
- pl1std.in, 16
- PLIRETC built-in subroutine, 69
- PLISRTA, 63
 - example, 67, 68, 72
- PLISRTB, 63
 - example, 69
- PLISRTC, 63

- PLISRTD, 63
 - example, 68, 72
- Pointers, passing as arguments, 96
- Preprocessor, 8
 - built-in functions, 11
 - procedures, 9, 11
 - statements, 10
 - variables, 8
- Procedure Calling Standard support, 94
- Procedures
 - calling, 100
 - passing as arguments, 98
 - preprocessor, 9
- Process permanent files, 20
- Program, executable, 3
- Prompt
 - with GET statement, 56
- PROMPT option, 56
- Protection files, 53, 53
- PURGE_TYPE_AHEAD option, 57
- PUT statement, 21

R

- Random file access, 26
- Reading files, 27, 32, 32
- RECORD condition, 22
- Record Management Services (RMS), condition values, 24
- Records
 - cell number, 26
 - format, 27
 - fixed-length, 28, 79
 - variable-length, 28
 - identification, accessing records by, 27
 - read into varying strings, 28
 - record files, 25
 - record I/O, 25
 - specifying size, 28, 31
- RECORD_ID option, 27, 57
- RECORD_ID_ACCESS ENVIRONMENT option, 27, 48
- RECORD_ID_TO option, 27, 58
- Reference, passing by, 95
 - ANY attribute, 23, 95

Relative files, 30, 31, 32, 32

- creating, 31, 31
- examples, 30
- handling errors in, 33
- keys of, 26
- ONKEY built-in function, 93
- populating, 32
- updating, 32

RESIGNAL built-in subroutine, 37, 76, 78, 79

REVERT statement, 90

REWIND built-in subroutine, 60

Routines

- non-PL/I, 95

Run-time

- errors, 15
- libraries, 7
 - search path, 7
 - search path with -L, 8
- messages, 108

Running a PL/I program, 15

S

SCALARVARYING ENVIRONMENT option, 49

Scope

- in sort routines, 69

Search order for ON-units, 33, 85, 86, 87

Sequential file access, 26

SET PROTECTION command, 54

Severity

- of compiler errors, 14
- of conditions, 92
- of resigned condition, 86

SHARED_READ ENVIRONMENT option, 50

Signal condition

- user-defined values, 76
- values, 91

Signal values, 82

signal(3), 45, 81

SKIPREC (sort option), 66

sort command, 62

Sorting

- calling the entry points, 63
- compatibility, 62

data format, 65

examples of calls, 72

file prefix, 68

file-prefix, 64

format of call, 64

input routines, 69

input-routine, 64

maximum-storage, 64

message-level, 65, 69

output routines, 69

output-routine, 64

PLIRETC built-in subroutine, 69

record length, 67

record type, 66

record-specification, 64

return-code, 64, 68

scope of variables, 69

sort command, 62

sort field length, 65

sort order, 66

sort-specification, 64, 65

starting point, 65

storage, 67

Source file

- compiler-generated line numbers, 15
- creating, 2

Standard input, 20

Standard output, 20

stdin, 20

stdout, 20

STOP statement in ON-unit, 81

Storage (for sorting), 67

STORAGE condition, 92

Stream files, 29

- appending records to, 29

- appending to, 30

- creating, 29

- input/output processing, 29

STRINGRANGE condition, 92

Structures

- using as file layout, 32

Subroutines for file-handling, 60

SUPERSEDE ENVIRONMENT option, 30, 50

SYSIN, 91
defaults for, 21

SYSPRINT
defaults for, 21
System routines, 94

T

TEMPORARY ENVIRONMENT option, 51
Terminal input
displaying prompting message, 56
suppressing display, 56
TITLE option, 19, 20
Traceback, run-time file errors, 24
TRANSMIT condition, 22
TRUNCATE attribute, 99
Type-ahead buffer, purging, 57

U

ucode, 3
UNDEFINEDFILE condition, 20, 22, 92, 93
ENVIRONMENT option conflicts, 36
invalid file specifications, 20
UNDERFLOW condition, 92
Unwind, 80, 80, 89
Updating relative files, 32
User identification code, specifying, 53

V

V, VB, VS, and VSB ENVIRONMENT options, 52
VALUE attribute, 98
parameter descriptor, 97
using with ANY, 98
Value, passing by, 97
Variable-length records, 28
Variables
environment, 20
specifying file title with, 20
file, 18, 19
file, specifying title with, 19
preprocessor, 8
using in signals, 82
Varying character strings, 28
VAXCONDITION condition, 76, 92

calling during unwind, 89
located in search for ON-units, 34, 86, 87
Version numbers, default, 21

W

WORLD_PROTECTION ENVIRONMENT option, 52, 53
Writing files, 32, 32

Z

ZERODIVIDE condition, 92