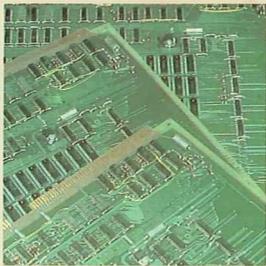
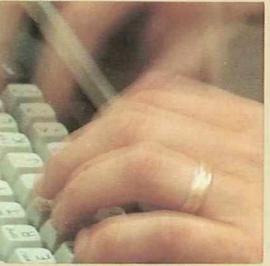
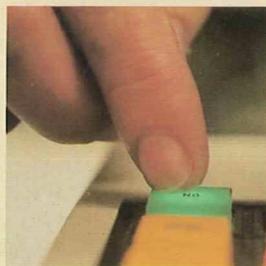
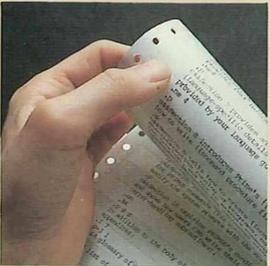


Prime Computer, Inc.

PRIME

DOC5041-1LA PL/I Reference Guide



PL/I Reference Guide

First Edition

by

**John J. Xenakis
and
Camilla B. Haase**

Updated for Translator Family
Release T1.0-21.0

by

Kim M. Seward

This guide documents the software operation of the Prime Computer and its supporting systems and utilities at Translator Family Release T1.0-21.0.

**Prime Computer, Inc.
Prime Park
Natick, Massachusetts 01760**

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

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

Copyright © 1988 by Prime Computer, Inc. All rights reserved.

PRIME, PRIME, PRIMOS, and the PRIME logo are registered trademarks of Prime Computer, Inc. PERFORMER, PRIME/SNA, PRIME TIMER, PRIMECALC, PRIMELINK, PRIMENET, PRIMWORD, PRODUCER, PST 100, PT25, PT45, PT65, PT200, FW153, FW200, FW250, RINGNET, 50 Series, 400, 750, 850, 2250, 2350, 2450, 2550, 2650, 2655, 2755, 6350, 6550, 9650, 9655, 9750, 9755, 9950, 9955, and 9955II are trademarks of Prime Computer, Inc.

PRINTING HISTORY

First Edition (DOC5041-11A) January 1986 for Release 1.0
Update 1 (UPD5041-11A) January 1988 for T1.0-21.0

CREDITS

Editorial: Bill Modlin, Thelma Henner
Project Support: Margaret Taft, Camilla Haase
Illustration: Marlene Bober
Illustration Support: Anna Spoerri
Graphic Support: Mingling Chang
Document Preparation: Julie Cyphers, Mary Mixon
Production: Judy Gordon

HOW TO ORDER TECHNICAL DOCUMENTS

To order copies of documents, or to obtain a catalog and price list:

United States Customers

Call Prime Telemarketing,
toll free, at 1-800-343-2533,
Monday through Friday,
8:30 a.m. to 5:00 p.m. (EST).

International

Contact your local Prime
subsidiary or distributor.

CUSTOMER SUPPORT

Prime provides the following toll-free numbers for customers in the United States needing service:

1-800-322-2838 (within Massachusetts)	1-800-541-8888 (within Alaska)
1-800-343-2320 (within other states)	1-800-651-1313 (within Hawaii)

For other locations, contact your Prime representative.

SURVEYS AND CORRESPONDENCE

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Contents

ABOUT THIS BOOK	xi
PART I -- OVERVIEW OF PRIME PL/I	
1 INTRODUCTION	
The PL/I Language	1-1
Restrictions on PL/I Programs	1-2
Interface to Other Languages	1-2
PL/I and the Editor	1-3
PL/I Under PRIMOS	1-4
Program Environments	1-5
PL/I and Prime Utilities	1-6
The PRIMOS Condition-Handling Mechanism	1-8
2 USING THE PL/I COMPILER	
Compiling a PL/I Program	2-1
Compiler Options	2-3
3 LINKING AND EXECUTING PL/I	
Introduction	3-1
How to Use BIND	3-1
Basic Linking Commands	3-3
Running Your Program	3-3
PART II -- PRIME PL/I LANGUAGE REFERENCE	
4 THE PL/I LANGUAGE	
Simple PL/I Programs	4-1
Elements of a PL/I Program	4-7
Expressions	4-11
Flow of Control With IF, DO, and GO TO	4-13
Numeric Data Types	4-22
Built-in Functions	4-28
CHARACTER String Data Type	4-32
Operations on CHARACTER Strings	4-35
Arrays and Structures	4-42

Input/Output	4-46
Other Features of the PL/I Language	4-49
5 DATA TYPES AND DATA ATTRIBUTES	
Data Types: Introduction	5-1
Arithmetic Data Types: Introduction	5-4
String Data Types: Introduction	5-18
Pictured Data Types: Introduction	5-29
Pictured-String	5-29
Pictured-Numeric	5-31
Arrays and Structures	5-56
The ALIGNED and UNALIGNED Attributes	5-66
The DEFINED Attribute	5-67
The LIKE Attribute	5-71
The INITIAL Attribute	5-72
The DEFAULT Statement	5-75
6 EVALUATING EXPRESSIONS	
Expressions, Data Conversions, and Aggregate Promotions	6-1
Forming Expressions	6-2
Scalar Targets and Data Conversions	6-9
PL/I Expression Operators	6-25
Scalar Conversion Rules for Computational Data Types	6-38
Aggregate Targets and Promotion	6-43
7 STORAGE MANAGEMENT	
Types of Storage	7-2
Techniques for Overlaying Storage	7-20
Extent Expressions and INITIAL Attribute	7-26
Internal and External Scope Attributes	7-30
Named Constants and Noncomputational Variables	7-31
Advanced Programming Option: POINTER OPTIONS(SHORT)	7-32

8 SUBROUTINE AND FUNCTION PROCEDURES

Procedures	8-1
Subroutine Procedures	8-3
Function Procedures	8-11
Summary of Differences Between Subroutine and Function Procedures	8-21
Relation Between Arguments and Parameters	8-21
External Procedures	8-31
Recursive Procedures	8-37
Generic Entry Names	8-39
ENTRY Variables	8-42
Advanced Programming Options: SHORTCALL and NONQUICK	8-44
Summary of Procedure Rules	8-45

9 PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

PL/I Program Block Structures	9-1
The DECLARE Statement	9-6
Types of Declarations	9-13
Scope of a Declaration	9-21
Resolving References	9-27

10 FLOW OF CONTROL

The IF Statement	10-1
The DO Statement	10-4
The GO TO Statement	10-22
The LEAVE Statement -- Prime Extension	10-24
The SELECT Statement -- Prime Extension	10-25
PL/I Program Blocks	10-27
Static and Dynamic Program Block Structure	10-29
Compiler-directing Statements	10-53

11 STREAM INPUT/OUTPUT

Introduction to the PUT Statement	11-2
Introduction to the GET Statement	11-13
PUT and GET to Files and Devices	11-24
DECLARE, OPEN, and TITLE	11-25
STREAM Input/Output Specifications	11-30
Establishing Data Items	11-42

Matching Data Values to Format Items	11-43
Detailed Specifications for the PUT Statement	11-46
Detailed Specifications for the GET Statement	11-66
12 RECORD INPUT/OUTPUT	
Concepts of RECORD Input/Output	12-1
Sequential RECORD Input/Output	12-5
Direct Access with DAM Files	12-10
Direct Access with MIDASPLUS Files	12-14
RECORD Input/Output in Locate Mode	12-17
File Attributes, Attribute Merging, and the OPEN Statement	12-20
Input/Output on Conditions and Built-in Functions	12-32
FILE Variables and Functions That Return FILE Values	12-33
13 PL/I CONDITION HANDLING	
The ON Statement	13-2
Enabling Conditions With Condition Prefixes	13-13
The REVERT Statement	13-18
The SIGNAL Statement	13-19
The CONDITION Condition	13-20
The SNAP Option	13-20
List of Conditions	13-24
Built-in Functions Related to On-Units	13-40
14 BUILT-IN FUNCTIONS AND PSEUDOVARIBLES	
Arguments to Built-in Functions	14-1
Classification and Summary of Built-in Functions	14-6
Complete List of Built-in Functions	14-13
The Use of Pseudovariables	14-95

APPENDICES

A	PL/I KEYWORDS	A-1
B	THE PRIME EXTENDED CHARACTER SET	
	Specifying Prime ECS Characters	B-2
	Special Meanings of Prime ECS Characters	B-5
	PL/I Programming Considerations	B-5
	Prime Extended Character Set Table	B-6
C	DATA FORMATS	
	Overview	C-1
	FIXED BINARY Data	C-2
	FIXED DECIMAL Data	C-2
	FLOAT BINARY Data	C-3
	FLOAT DECIMAL Data	C-3
	COMPLEX FIXED BINARY Data	C-4
	COMPLEX FIXED DECIMAL Data	C-4
	COMPLEX FLOAT BINARY Data	C-5
	COMPLEX FLOAT DECIMAL Data	C-5
	PICTURE Data	C-6
	CHARACTER Data	C-6
	CHARACTER VARYING Data	C-7
	BIT Data	C-7
	BIT VARYING Data	C-7
	POINTER Data	C-8
	POINTER OPTIONS (SHORT)	C-8
	LABEL Data	C-8
	ENTRY Data	C-9
	FILE Data	C-10
	Arrays	C-10
D	FUNCTION RETURN CONVENTIONS AND STACK FRAME FORMAT	
	Locations of Returned Function Values	D-1
	Stack Frame Format	D-2

E	DIFFERENCES AMONG ANS, IBM, AND PRIME PL/I	
	Prime Extensions to the ANSI Standard	E-1
	ANS Feature Not Supported in Prime PL/I	E-2
	IBM Features Not Supported in Prime PL/I	E-2
F	ONCODE VALUES	F-1
G	GLOSSARY OF PL/I TERMS	G-1
H	USE OF FORMS WITH PL/I	H-1
I	USING SEG	I-1
	INDEX	X-1

About This Book

This book is a programmer's guide to the PL/I language as implemented on Prime systems. You are expected to be familiar with some high-level language and with programming in general, but not necessarily with PL/I or Prime computers. If you need additional background in programming techniques or PL/I, consult an appropriate textbook, such as:

Conway, Richard, and Gries, David, An Introduction to Programming: A Structured Approach Using PL/I and PL/C, Little, Brown, 1978.

Hughes, Joan K., PL/I Structured Programming, John Wiley & Sons, Inc., 1979.

Pollack, Seymour V., and Sterling, Theodore C., A Guide to PL/I and Structured Programming, Holt, Rinehart, and Winston, 1980.

NEW FEATURES

The following features are new to Prime PL/I at Translator Family Release T1.0-21.0:

1. Search rules for %INCLUDE files. See Chapter 10.
2. The Prime Extended Character Set and the -EXTENDED_CHARACTER_SET and -NO_EXTENDED_CHARACTER_SET compiler options. See Appendix B and Chapter 2.

3. The `-STRINGSIZE` and `-NO_STRINGSIZE` compiler options. See Chapter 2 and Chapter 13.

HOW TO USE THIS BOOK

The following is a brief chapter-by-chapter description of the contents of this book.

Part I -- Overview of Prime PL/I

- Chapter 1 introduces PL/I and its implementation on Prime computers.
- Chapter 2 provides information on the use of the PL/I compiler, including compiler options.
- Chapter 3 provides information on linking and executing programs using Prime's BIND utility.

Part II -- Prime PL/I Language Reference

- Chapter 4 provides an overview of the PL/I language.
- Chapter 5 describes PL/I data elements.
- Chapter 6 describes PL/I expressions, data type conversions, and aggregate data structures.
- Chapter 7 describes PL/I storage management capabilities and the types of storage available.
- Chapter 8 explains the use of subroutines and functions and the differences between them.
- Chapter 9 describes PL/I program blocks and explains the rules of scoping in declarations.
- Chapter 10 describes the flow of control of a PL/I program and briefly discusses compiler-directing statements.
- Chapter 11 describes how to input and output data on the terminal.
- Chapter 12 describes how to input and output data by using files and devices.
- Chapter 13 describes PL/I condition handling.

- Chapter 14 describes PL/I's built-in functions and pseudovariables.

Appendixes

- Appendix A lists all PL/I keywords.
- Appendix B describes and lists the Prime Extended Character Set. |
- Appendix C describes the internal representation of PL/I data types.
- Appendix D describes PL/I function return conventions and the stack frame format.
- Appendix E summarizes the differences between Prime PL/I and ANS PL/I and between Prime PL/I and IBM PL/I.
- Appendix F lists PL/I error codes and their meanings.
- Appendix G provides a glossary of PL/I terms.
- Appendix H briefly describes the use of Prime's FORMS utility with PL/I.
- Appendix I summarizes the use of Prime's older linking utility, SEG.

RELATED DOCUMENTS

In addition to the PL/I Reference Guide, there are several books describing other Prime utilities that will help you with your programming on Prime equipment. These documents are:

Prime User's Guide (DOC4130-4LA)

Prime PL/I Conversion Guide (DOC5769-1LA)

EMACS Primer (IDR6107-183) and EMACS Reference Guide (DOC5026-2LA)

New User's Guide to EDITOR and RUNOFF (DOC3104-4LA)

Programmer's Guide to BIND and EPFs (DOC8691-1LA)

SEG and LOAD Reference Guide (DOC3524-192L)

Subroutines Reference Guide

Volume I (DOC10080-2LA)

Volume II (DOC10081-1LA)

Volume III (DOC10082-1LA)

Volume IV (DOC10083-1LA)

Source Level Debugger User's Guide (DOC4033-193L)

Guide to Prime User Documents (DOC6138-5PA)

PRIMOS Commands Programmer's Companion (FDR3250-192)

Other Sources of Information

In addition to the documents listed above, please consider the following sources when looking for information about Prime PL/I:

- The Software Release Document released at each software revision. This document contains a summary of new features and changes in Prime's user software.
- Prime's online HELP files. Information on PRIMOS commands is displayed at your terminal. A cumulative list of manuals, updates, and other material is also included.

PRIME DOCUMENTATION CONVENTIONS

The following conventions are used in command formats, statement formats, and in examples throughout this document. Examples illustrate the uses of these commands and statements in typical applications. Terminal input may be entered in either uppercase or lowercase.

<u>Convention</u>	<u>Explanation</u>	<u>Example</u>
UPPERCASE	In command formats, words in uppercase indicate the actual names of commands, statements, and keywords. They can be entered in either uppercase or lowercase.	SLIST
lowercase	In command formats, words in lowercase indicate items for which the user must substitute a suitable value.	LOGIN user-id

<u>Convention</u>	<u>Explanation</u>	<u>Example</u>
Abbreviations	If a command or statement has an abbreviation, it is indicated by underlining. In cases where the command or directive itself contains an underscore, the abbreviation is shown below the full name, and the name and abbreviation are placed within braces.	<u>LOGOUT</u> SET_QUOTA SQ
<u>Underlining</u> in examples	In examples, user input is underlined but system prompts and output are not.	OK, <u>RESUME MY_PROG</u> This is the output of MY_PROG.CPL OK,
Brackets	Brackets enclose a list of two or more optional items. Choose none, one, or more of these items.	SPOOL [-LIST -CANCEL]
Braces	Braces enclose a list of items. Choose one and only one of these items.	CLOSE { filename } ALL }
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.	item-x[,item-y]...
Parentheses ()	In command or statement formats, parentheses must be entered exactly as shown.	DIM array (row,col)
Hyphen -	Wherever a hyphen appears as the first letter of an option, it is a required part of that option.	SPOOL -LIST

PART I

Overview of Prime PL/I



1

Introduction

THE PL/I LANGUAGE

PL/I is a comprehensive general-purpose programming language combining the best features of several other languages, including FORTRAN, COBOL, and ALGOL. PL/I provides more powerful programming tools and methods than any other language currently available. It is defined in the American National Standards Institute (ANSI) document X3.53-1976.

Prime's PL/I is completely compatible with Prime's PL/I Subset G (invoked by the command PLIG). Any extensions available for Subset G are also available for PL/I. You can recompile programs written in Subset G with no changes using PL/I.

Differences Between Prime PL/I and Standard PL/I

Appendix E contains a detailed comparison of Prime PL/I and ANSI Standard PL/I. Differences between Prime PL/I and IBM PL/I are also listed there.

Implementation-defined Features of PL/I

The ANSI standard for PL/I does not specify every detail of the language. Certain features that are inherently dependent on the particular computer system used are designated in the standard as implementation-defined. Each computer manufacturer sets its own standard for such features.

A general description of each implementation-defined feature is given in the appropriate chapter of this guide. Specific details on Prime's choices for each such feature are contained in Appendix C.

Programs that may have to run under a non-Prime version of PL/I should be written to be minimally dependent on implementation-defined features.

Compatibility of Prime PL/I Subset G

If PL/I-G programs are recompiled with full PL/I, you should load and execute them using Rev. 19.4 or higher PRIMOS.

RESTRICTIONS ON PL/I PROGRAMS

The segmented nature of the Prime virtual memory system imposes a few machine restrictions on PL/I programs. None of these restrictions is contrary to the ANSI standard or need interfere with program design.

- The executable code (exclusive of data storage) for a compilation unit may not occupy more than one segment (128K bytes). For additional program space, break out procedures and make them separate compilation units.
- No program may have more than one segment of local static storage. For additional storage, make some of the data static external.
- No program unit may have more than one segment of dynamic storage. Any additional storage must be made static.
- No data item in a static external aggregate may be split across the boundary between two segments. When laying out a static external aggregate, use the information on storage formats in Appendix C to insure compliance with this rule.

INTERFACE TO OTHER LANGUAGES

Since all Prime high-level languages are alike at the object code level, and since all use the same calling conventions, object modules

produced by the PL/I compiler can reference or be referenced by modules produced by the FORTRAN IV, FORTRAN 77, COBOL, Pascal, and C compilers. You must observe certain restrictions when a PL/I object module interfaces one compiled from another language.

- All I/O routines must be written in the same language.
- There must be no conflict of data types for variables being passed as arguments. For example, FIXED BINARY in PL/I should be declared as INTEGER in FORTRAN 77. See Appendix C for a description of PL/I data storage formats.
- Modules compiled in 64V or 32I mode cannot reference or be referenced by modules compiled in any R mode. Modules in 64V or 32I may reference each other if they are otherwise compatible.
- A PL/I program cannot reference a FORTRAN complex-valued function.
- A label passed to a Prime FORTRAN IV (FTN) subroutine as an alternate-return specifier must identify a statement in the same block that contains the subroutine call.

You may use a PL/I static external structure to reference a FORTRAN or PMA common block having the same name as the structure. Take care that the data items in the structure and block correspond appropriately.

PL/I object modules can also interface with PMA (Prime Macro Assembler) routines. See the Assembly Language Programmer's Guide.

You may input any data file to PL/I, providing it is written in either ASCII compressed or binary form.

PL/I AND THE EDITOR

PL/I source code can be entered into Prime systems using the system line editor, called EDITOR.

The characters up-arrow (^) and semicolon (;) have special meanings for the EDITOR that conflict with their uses in PL/I. The ^ is the EDITOR's escape character and PL/I's NOT character, while the ; is the EDITOR's carriage return and PL/I's statement delimiter. A conflict arises whenever you attempt to enter either of these characters into a PL/I source program using the EDITOR.

The EDITOR functions of ^ and ; can be transferred to other symbols for the duration of an EDITOR session by using the EDITOR's SYMBOL command. While in EDIT mode, type:

```
SY SEMICO a
SY ESCAPE b
```

PL/I Reference Guide

where a and b must be single, currently non-special characters. (The @ and ~ characters may be useful.) The character a replaces the semicolon as a carriage return, and b replaces the up-arrow as an escape character. The semicolon and up-arrow are thereby freed for ordinary use.

The semicolon may also be freed by typing in MODE NOSEMI in EDIT mode. The up-arrow may be entered as the double symbol ^^ without changing its function as an escape character.

For more information, see the New User's Guide to EDITOR and RUNOFF.

More permanent solutions to this conflict are available through your Prime field analyst. The EMACS capability is available as a separately priced product.

PL/I UNDER PRIMOS

Implementation

Prime's PL/I runs on all Prime models. It operates under PRIMOS, Prime's operating system. Code generated by the PL/I compiler is the same for all Prime processor models.

The maximum code size for a PL/I program is 64K.

Prime's processors execute an extended set of instructions directly, including decimal arithmetic and character edits. They maximize execution time efficiency better than processors that only recognize the code as an unimplemented instruction trap and automatically substitute an equivalent software routine. Addressing modes and the Prime instruction set hardware are presented in the System Architecture Guide (DOC9473-11A), the Instruction Sets Guide (DOC9474-11A), and the Assembly Language Programmer's Guide (FDR3059-101A).

Operating Environment

Only one version of PRIMOS exists for all Prime models. It features paged and segmented virtual memory management. The system is based on demand paging from disk with 2048 bytes per page. A page-sharing feature reduces overhead time. The system thus supplies paging requirements for the application program immediately and automatically. For example, several users may share one copy of the EDITOR to enter or modify their programs, rather than having multiple copies.

PROGRAM ENVIRONMENTS

Under PRIMOS, PL/I programs may be run in one of three environments:

- Interactive
- Phantom user
- Batch job

Interactive

All phases of PL/I compilation can be handled through interactive terminals. Therefore, you can enter and modify source programs directly at a terminal. You can create, edit, compile, list, debug, execute, and save a program in a single interactive session.

Program execution is initiated directly by you. Programs run in real time and are associated with a terminal. You can display program output, as well as error messages, at the terminal.

Major interactive uses are

- Program development
- Programs requiring short execution time
- Data entry programs, such as order entry or payroll
- Interactive programs, such as the EDITOR

Phantom User

The phantom environment allows programs to be executed while not associated with a terminal. This frees the terminal for other uses. Phantom users are programs that accept input from a command file instead of a terminal; output directed to a terminal is either ignored or directed to a file.

Major uses of phantoms are

- Programs requiring long execution time, such as sorts
- Certain system utilities, such as line printer spooler
- Any program, if the terminal should be free for another use

For more information on command files and phantom users, see the Prime User's Guide.

Batch Job

Since the number of phantom users on a system is limited, phantoms are not always available. The batch environment allows users to submit noninteractive command files as batch jobs at any time. The Batch Monitor (itself a phantom) queues these jobs and runs them, one to six at a time, as phantoms become free.

For more information on command files and batch processing, see the Prime User's Guide.

PL/I AND PRIME UTILITIES

Prime offers three major utility systems for use by Prime programmers. These are

- Multiple Index Data Access System (MIDASPLUS)
- Forms Management System (FORMS)
- The Source Level Debugger

For complete information on any of these utilities, see the appropriate reference guide. Below are brief descriptions of MIDASPLUS, FORMS, and the Debugger.

Multiple Index Data Access System (MIDASPLUS)

MIDASPLUS is a system of interactive utilities and high-level subroutines enabling the use of index-sequential and direct-access data files at the applications level. MIDASPLUS automatically handles indexes, keys, pointers, and the rest of the file structure. Major advantages of MIDASPLUS are

- Large data files
- Efficient search techniques
- Rapid data access
- Compatibility with existing Prime file structures
- Ease of building files
- Multiple user access to files
- Partial or full file deletion utility

PL/I interfaces with MIDASPLUS through the use of MIDASPLUS subroutine calls invoked by the OPEN, WRITE, REWRITE, and READ statements for a file declared as KEYED SEQUENTIAL with the -DAM option. See Chapter 12 of this manual and the MIDASPLUS User's Guide (DOC9244-1LA). Since MIDASPLUS subroutines are written in Prime FORTRAN IV, the restrictions mentioned earlier in this chapter under INTERFACE TO OTHER LANGUAGES apply.

FORMS Management System (FORMS)

The Prime FORMS Management System (FORMS) provides a convenient method of defining a form in a language specifically designed for such a purpose. These forms may then be implemented by any applications program that uses Prime's Input/Output Control System (IOCS), including programs written in PL/I. Applications programs communicate with FORMS through input/output statements native to the host language. Programs that currently run in an interactive mode can easily be converted to use FORMS.

FORMS allows cataloging and maintenance of form definitions available within the computer system. For use within an applications program, all form definitions reside within a centralized directory in the system. This directory, under control of the System Administrator, may be easily changed, allowing the addition, modification, or deletion of form definitions.

The interface of PL/I with FORMS is identical to that of Prime FORTRAN IV.

See the FORMS Programmer's Guide, or Appendix H of this manual.

The Source Level Debugger

Prime's powerful interactive debugging tool, the Source Level Debugger, may be obtained from Prime as a separately priced item. Use of the Debugger can greatly expedite and simplify the debugging process. Major features of the debugger enable you to

- Set both absolute and conditional breakpoints
- Request the execution of Debugger commands (action list) when a breakpoint occurs
- Execute the program step by step
- Call procedures, subroutines, or functions from Debugger command level
- Trace statement execution

- Trace selected variables, printing a message when their value changes
- Print or change the value of any variable
- Create Debugger macros that stand for two or more Debugger commands
- Print a subprogram call or return stack history (traceback)
- Examine the source file while executing within the Debugger, eliminating the need for hard-copy listings

See the Source Level Debugger User's Guide.

THE PRIMOS CONDITION-HANDLING MECHANISM

PRIMOS has two ways of reporting and dealing with errors: error codes and PRIMOS conditions.

When a PRIMOS subroutine is called, it returns an error code. This code must be tested by the calling program to establish that the subroutine has executed successfully.

Some errors cannot be dealt with by the return of an error code. For each such error, a PRIMOS condition exists. When the error occurs, the condition corresponding to the error is raised.

When a condition is raised, PRIMOS activates the condition-handling mechanism. The condition handler notes what condition exists, then calls an error-handling routine known as an on-unit to deal with the error that has occurred.

PRIMOS supplies a default on-unit that handles all conditions. You can specify an individual response to a condition by supplying an on-unit. When a condition occurs for which a programmer-supplied on-unit exists, the actions specified in the on-unit are taken, rather than those specified in the PRIMOS default on-unit.

Information on the system default on-unit and the method for substituting programmer-supplied on-units is contained in the Prime User's Guide. For complete information on the condition handler, see the Subroutines Reference Guide.

2

Using the PL/I Compiler

Prime's PL/I compiler accepts a source program meeting the PL/I standard. It translates the statements in the source program into an object (binary) module that contains the machine code needed to link and execute the program. It can also output a source listing, error and statistics information, and various messages. Errors are reported at the terminal as the compiler detects them.

This chapter describes:

- How to compile PL/I programs
- How to specify options to the compiler
- Compiler error messages
- Compiler options

COMPILING A PL/I PROGRAM

After you have entered your source program into the system using ED or EMACS, and have named the program with a .PLI suffix, you are ready to invoke the PL/I compiler.

Invoking and Specifying Options to the Compiler

To invoke the PL/I compiler from the PRIMOS command level, use the PLI command:

```
PLI pathname [-option_1] [-option_2] . . . [-option_n]
```

pathname is the pathname of the PL/I source program to be compiled. If the source program has the suffix .PLI, pathname need only include the part to the left of the period.

option is a PL/I compiler option. These options provide information and input while you compile, link, and execute your program.

All compiler options begin with a hyphen (-). For example,

```
OK, PLI TEST1 -RANGE -DEBUG -LISTING
```

causes TEST1.PL1 to be compiled with the options given. You may specify more than one option on the command line, in any order. However, if you issue conflicting options, an error message results.

Compiler Error Messages

During compilation, the compiler outputs an error message each time it encounters an error in your program. The error messages, which are self-explanatory, assist you in finding and correcting the errors in your program. For every error found, the compiler displays information about where the error occurred and the level of severity:

```
ERROR xxx SEVERITY y BEGINNING ON LINE zzz  
explanation of message
```

xxx Error code

y Level of severity

zzz Line number where error begins

explanation Description of the error and possible remedies

Errors are classified into four levels depending on the severity of the error:

<u>Severity</u>	<u>Meaning</u>
1	Warning -- a recoverable error, object file produced.
2	Recoverable -- the compiler has supplied defaults or a conversion; object file produced.
3	Nonrecoverable -- object file not produced.
4	Error that immediately aborts the compilation.

After the compilation process is complete, the compiler prints an end-of-compilation message at the terminal. Its format is:

```
xxxx ERRORS [PL1 Rev. 19.4]
MAX SEVERITY IS y
```

xxxx is the number of compilation errors; 0000 indicates a successful compilation. If there are no errors, the second line of the message does not appear.

After compilation, control returns to the PRIMOS level. The PRIMOS error prompt appears after a compilation that results in error messages of severity levels greater than 2. Any PRIMOS command that may be entered after OK may also be entered after the error prompt. (The default error prompt, which can be changed by users, is ER!)

COMPILER OPTIONS

This section discusses the options available with the PL/I compiler. Most of the options come in pairs, which act as switches to enable or disable a particular action. The Prime-supplied defaults are indicated by an asterisk. These defaults can be changed by your System Administrator. Some options require an argument to the option specification. The argument follows the option and is not preceded by a hyphen.

At the end of this section, Table 2-1 lists a summary of compiler options and abbreviations.

▶ *-64V

The -64V option generates 64V-mode code, which is a segmented virtual addressing mode for 32-bit machines.

▶ *-ALLOW_PRECONNECTION / -NO_ALLOW_PRECONNECTION

Abbreviation: -APRE / -NAPRE

The -ALLOW_PRECONNECTION option allows for the preconnection of a listing output to a preopened file unit 2, or of a binary output to a preopened file unit 3. When files have been preconnected, the compiler displays a message indicating that preconnection has occurred.

When the -NO_ALLOW_PRECONNECTION option is used, PL/I always opens and closes the listing and binary files (and uses dynamic file units).

▶ -BIG / *-NO_BIG

Abbreviation: -BIG / -NBIG

-BIG generates segment-spanning code for aggregates (arrays or structures) larger than one segment, when such aggregates are passed as parameters or referenced in a subprogram. If -BIG is specified, a BASED or PARAMETER aggregate can become associated with any aggregate, whether or not a segment boundary is crossed.

-NOBIG specifies that a BASED or PARAMETER aggregate can become associated only with an aggregate that does not cross a segment boundary.

▶ *-BINARY [pathname] / -NO_BINARY

Abbreviation: -B / -NB

The -BINARY option produces an object (binary) file with the name source-program.BIN. To write the object code to a different file, use the -BINARY option followed by pathname.

-NO_BINARY specifies that no binary object file is to be produced. Use this option when only a syntax check or listing is desired.

► `*-COPY / -NO_COPY`

Abbreviation: `-COP / -NOOP`

`-COPY` causes the compiler to copy constants before calling subprocedures, that is, to pass by value, so that when the new copy is changed by a procedure, the corresponding value in the calling routine is not changed.

`-NO_COPY` allows the programmer to suppress the copying of constants into temporary variables for procedure calls. This feature must be coded properly in the called procedure, for if the called routine changes the value of one of its parameters which was passed as a constant, the value of that constant in the calling program will be changed, causing subsequent references to that constant to use the wrong value. Use of the `-NO_COPY` option can save on the amount of executable code generated.

► `-DEBUG / *-NO_DEBUG`

Abbreviation: `-DBG / -NDBG`

`-DEBUG` controls generation of code for the debugger. The object file is modified so that it will run under the Source Level Debugger. Execution time increases, and the code generated is not optimized. Use of the debugger on programs with external procedures is not supported.

`-NO_DEBUG` causes no debugger code to be generated.

The Source Level Debugger is a separately priced product. For full information, see the Source Level Debugger User's Guide.

► `-ERRLIST / *-NO_ERRLIST`

Abbreviation: `-ERRL / -NERRL`

`-ERRLIST` produces a listing file (see below under `-LISTING`) that contains only error messages. If both `-LISTING` and `-ERRLIST` are specified, `-ERRLIST` takes precedence.

`-NO_ERRLIST` has no effect; if both `-LISTING` and `-NO_ERRLIST` are specified, the listing file still includes the error messages.

▶ `*-ERRITY / -NO_ERRITY`

Abbreviation: `-ERRT / -NERRT`

`-ERRITY` prints error messages at the terminal during compilation.

`-NO_ERRITY` suppresses listing of errors on the terminal. They are still included in the source listing file, if there is one.

▶ `-EXPLIST / *-NO_EXPLIST` (Implies `-LISTING`)

Abbreviation: `-EXP / -NEXP`

`-EXPLIST` inserts a pseudo-assembly code listing into the source listing. Each statement in the source is followed by the pseudo-PMA (Prime Macro Assembler) statements into which it was compiled. For information on PMA, see the Assembly Language Programmer's Guide.

`-NO_EXPLIST`, if specified with `-LISTING`, causes no assembler statements to be printed in the listing.

▶ `-EXTENDED_CHARACTER_SET / *-NO_EXTENDED_CHARACTER_SET`

Abbreviation: `-ECS / -NECS`

The `-EXTENDED_CHARACTER_SET` option causes the `COLLATE` built-in function to return a 256-character string, and causes the `LOW` built-in function to return the character with numeric value zero. (See Chapter 14 for information about these functions. The Prime Extended Character Set is discussed in Appendix B.)

`-NO_EXTENDED_CHARACTER_SET` makes `COLLATE` return a 128-character string, and makes `LOW` return the character with decimal value 128 (200 octal).

▶ `-FRN / *-NO_FRN`

Abbreviation: `-FRN / -NFRN`

The Floating-point Round option improves the accuracy of calculations involving single-precision real (`FLOAT BIN(23)`) numbers.

When the `-FRN` option has been given, all single-precision numbers are rounded each time they are moved from a register to main storage. The method of rounding is as follows: if the last bit of the mantissa is 1, add a 1 to the second-to-last bit, then set the last bit to 0. This rounding reduces loss of accuracy in the low-order bits when many calculations are performed on the same number.

The -FRN option does not affect double-precision real numbers (FLOAT BIN(47)). It causes a slight increase in execution time and should therefore be used only when maximum accuracy is a major consideration.

-NO_FRN causes no rounding to be performed.

▶ -FULL_HELP

Abbreviation: -FH

-FULL_HELP is similar to the -HELP option, except that in addition to the usage summary, a description of the meaning of each compiler option is given. The -HELP option is described below.

▶ -FULL_OPTIMIZE

Abbreviation: -FOPT

-FULL_OPTIMIZE ensures that the maximum amount of optimization available is used. A note in the listing file shows the current level of optimization implied by the use of this option.

▶ -HELP

Abbreviation: -H

-HELP produces information on using the PL/I compiler. The compiler displays a usage summary and a list of all options available.

▶ -INPUT pathname

Abbreviation: -I

-INPUT is an alternative way to specify the source file to be compiled, if you do not name the file immediately after the PLI command. pathname specifies the name of the source program. If pathname is TTY, input will come from the terminal. The pathname must not be designated more than once. -INPUT is identical to the -SOURCE option; see the discussion of that option for examples.

▶ -LCASE

Abbreviation: -LC

The -UPCASE and -LCASE options control mapping of lowercase to uppercase letters in a source program.

-LCASE distinguishes between lowercase and uppercase letters. System calls must be in uppercase. -UPCASE is the default.

▶ -LISTING

TTY
pathname
SPOOL

 / *-NO_LISTING

Abbreviation: -L / -NL

-LISTING causes the creation of a source listing file with the name source-program.LIST. The file ordinarily has four components: a list of compiler options selected (including defaults), the source code with line numbers, a map of data and procedure names, and compiler error messages. The following arguments may be used:

TTY The listing is displayed at the terminal.

SPOOL The listing is spooled directly to the line printer.
Default SPOOL arguments are in effect.

To write the listing to a specific file, use the -LISTING option followed by pathname.

-NO_LISTING causes no listing file to be created.

▶ -MAP / *-NO_MAP (Implies -LISTING)

Abbreviation: -MA / -NMA

-MAP produces a listing file that contains a reference map of data and procedure names. To get a full cross-reference of usage information for each symbolic name, use the -XREF option. -MAP by itself is identical to -LISTING by itself.

-NO_MAP, if specified with -LISTING, produces a listing file that includes only the source program and the error messages, without a variable reference map.

► `-MAPWIDE [decimal-integer]` (Implies `-MAP`)

Abbreviation: `-MAPW`

`-MAPWIDE` specifies the width in characters of the cross-reference map that appears in the listing file, as well as the width of the options list section that appears at the beginning of the listing file. The legal range of values for the decimal-integer argument is from 80 to 160 inclusive. If a listing file is being produced and `-MAPWIDE` is not specified, the default map width is 80. `-MAPWIDE` with no argument is equivalent to `-MAPWIDE 108`.

► `-MAXERRORS [decimal integer]`

Abbreviation: `-MAXE`

`-MAXERRORS` specifies the maximum number of compilation errors to be reported. If in a given compilation the specified maximum is reached, then a severity 4 error message is issued and the compilation is aborted. The number of errors that can be reported can range from 1 to 32767. If `-MAXERRORS` is not specified, the default maximum number of errors to report is 100; if `-MAXERRORS` is specified without a decimal argument, the maximum number of errors to report is 32767.

► `-NESTING / *-NO_NESTING` (Implies `-LISTING`)

Abbreviation: `-NE / -NNE`

`-NESTING` includes logical control nesting level in the source listing. Each line in the source listing is printed with a number indicating how many PROCEDURE statements, BEGIN blocks, and DO groups contain the statement(s) on that line. This option is useful in tracing flow of control and in matching END statements with their corresponding DO, BEGIN, and PROCEDURE statements.

`-NO_NESTING`, if specified with `-LISTING`, produces a listing file that contains no nesting level numbers.

► `-OFFSET / *-NO_OFFSET` (Implies `-LISTING`)

Abbreviation: `-OFF / -NOFF`

`-OFFSET` appends an offset map to the source listing. For each statement in the source program, the offset map gives the offset in the object file of the first machine instruction generated for that statement.

`-NO_OFFSET`, if specified with `-LISTING`, causes no offset map to be appended to the listing file.

► `*-OPTIMIZE [decimal-integer]`

Abbreviation: `-OPT`

`-OPTIMIZE` controls the optimization phase of the compiler. Optimized code runs more efficiently than non-optimized code, but takes somewhat longer to compile. The decimal-integer that follows `-OPTIMIZE` specifies one of the following levels:

<u>Level</u>	<u>Meaning</u>
0	Perform no optimizations. Turns optimization off.
1	Code pattern replacement.
2	Common subexpression elimination. (Default value)
3	Loop invariant removal, code pattern elimination, and redundancy elimination. At this level, internally nested procedures are made <u>quick</u> , that is, called by a Jump To Subroutine instruction rather than a Procedure Call, if conditions allow. The condition under which a procedure is made quick is that it be called simply, that is, called from only one place. For example, procedure C can be quick if it is called only from procedure A. But if it is also called from procedure B, where B is a separate procedure from A, then C cannot be quick. The last section of Chapter 8 explains how to use the <code>NONQUICK</code> option to keep a particular procedure from being made quick.

Note

Each optimization level performs all the optimizations of the next lower level, plus those that are listed.

If you do not specify `-OPTIMIZE`, the default level is 2. `-OPTIMIZE` with no arguments also produces the default level. The level of optimization that you select is identified in the optimization note of the compiler's listing output file.

► -OVERFLOW / *-NO_OVERFLOW

Abbreviation: -OVF / -NOVF

-OVERFLOW enables the integer condition handling mechanism when a division by zero is encountered, or when integer arithmetic causes an integer to be larger than the data type for which it was declared. -OVERFLOW affects integer calculations only. It causes the FIXEDOVERFLOW condition to be raised at runtime if the result does not fit.

-NO_OVERFLOW disables the integer overflow condition.

► -PRODUCTION / *-NO_PRODUCTION

Abbreviation: -PROD / -NPROD

-PRODUCTION generates controlling code for the debugger. It is similar to -DEBUG, except that the code generated does not permit insertion of statement breakpoints or tracepoints, nor does it allow single-stepping through the program. Execution time increases less than when -DEBUG is specified.

-NO_PRODUCTION causes no production-type code to be generated.

► -RANGE / *-NO_RANGE

Abbreviation: -RA / -NRA

-RANGE controls error checking for out-of-bounds values of array subscripts and character substring indexes.

Error-checking code is inserted into the object file. If an array subscript or character substring index takes on a value outside the range specified when the referenced data item was declared, the ERROR condition is signalled. (Note that range checking decreases the efficiency of the generated code.)

-NO_RANGE causes no code to be generated to check for out-of-range values of subscripts and indexes.

► -SILENT [decimal-integer]

Abbreviation: -SI

-SILENT, when used with a decimal argument, suppresses the printing of error and warning messages of the severity you specify in decimal-integer. The error and warning messages are omitted from any

listing files generated. Severity levels are listed above in the section on Compiler Error Messages.

If `-SILENT` is not specified, a value of `-1` is assumed: all error messages appear. `-SILENT` with no argument is equivalent to `-SILENT 1`. The option header in the listing file (if any) shows the level of severity you specify in decimal-integer.

▶ `-SOURCE` pathname

`-SOURCE` is an alternative way to specify the source file to be compiled, if you do not name the file immediately after the PL1 command. pathname specifies the name of the source program. If pathname is `TTY`, input will come from the terminal. `-SOURCE` is identical to the `-INPUT` option. The following are equivalent:

```
PL1 pathname -RANGE -BIG
```

```
PL1 -RANGE -BIG -I pathname
```

```
PL1 -BIG -S pathname -RANGE
```

The pathname must not be designated more than once.

▶ `-SPACE`

`-SPACE` specifies that space reduction is to be given preference over speed in optimization consideration. This option is the opposite of `-TIME`, which favors speed over space in reducing the size of optimized code. `-TIME` is the default.

▶ `-STATISTICS` / `*-NO_STATISTICS`

Abbreviation: `-STAT` / `-NSTAT`

`-STATISTICS` displays a list of compilation statistics at the terminal after each phase of compilation. For each phase the list contains:

DISK	Number of reads and writes during the phase, excluding those needed to obtain the source file
SECONDS	Elapsed real time
SPACE	Internal buffer space used for symbol table, in 16K byte units

NODES	The number of symbol table nodes that the compiler is using in the program
PAGING	Disk I/O time
CPU	CPU time in seconds, followed by the clock time when the phase was completed

The `-NO_STATISTICS` option does not display compilation statistics at the terminal.

► `*-STORE_OWNER_FIELD / -NO_STORE_OWNER_FIELD`

Abbreviation: `-SOF / -NSOF`

`-STORE_OWNER_FIELD` stores the identity of the current program in a known place for use by traceback routines. This option is useful for debugging programs. Use of this option increases the size of the generated code and linkage and slightly degrades execution time of programs.

`-NO_STORE_OWNER_FIELD` omits this small code sequence for extremely time-critical programs.

► `*-STRINGSIZE / -NO_STRINGSIZE`

Abbreviation: `-STRZ / -NSTRZ`

`-STRINGSIZE` enables the `STRINGSIZE` condition for the entire compilation unit. PL/I raises the `STRINGSIZE` condition if either of the following circumstances occurs:

- In a `PUT EDIT` statement with the data format item `B[n](w)`, the resulting conversion contains more bits than the specified width w.
- A source string is longer than the maximum length of the assignment target.

`-NO_STRINGSIZE` disables the `STRINGSIZE` condition. Execution time is faster because code is not generated to check for string overflow at runtime.

Chapter 13, `PL/I Condition Handling`, contains more detailed information on the `STRINGSIZE` condition.

▶ *-TIME

-TIME specifies that speed is to be given preference over space reduction in optimization selection. This option is the opposite of -SPACE, which favors space over speed in reducing the size of optimized code.

▶ *-UPCASE

Abbreviation: -UP

The -UPCASE and -LCASE options control mapping of lowercase to uppercase letters in a source program.

-UPCASE treats all lowercase letters in the source as uppercase, except in character constants.

▶ -XREF / *-NO_XREF (Implies -LISTING)

Abbreviation: -XREF / -NXREF

-XREF appends a cross-reference to the source listing. A cross-reference gives, for every variable, the numbers of all lines on which the variable was referenced.

-NO_XREF does not generate a cross-reference listing.

Table 2-1
 Summary of Compiler Options and Abbreviations
 (Defaults are marked with asterisks.)

Option	Abbreviation	Significance
-64V *		Produce 64V mode code
-ALLOW_PRECONNECTION *	-APRE	Use preopened file
-NO_ALLOW_PRECONNECTION	-NAPRE	
-BIG		Allow boundary spanning
-NO_BIG *	-NBIG	
-BINARY *	-B	Create object file
-NO_BINARY	-NB	
-COPY *	-COP	Pass arguments by value
-NO_COPY	-NCOP	Pass arguments by reference
-DEBUG	-DBG	Generate debugger code
-NO_DEBUG *	-NDBG	
-ERRLIST	-ERRL	Produce an error file
-NO_ERRLIST *	-NERRL	
-ERRITY *	-ERRT	List errors on the terminal
-NO_ERRITY	-NERRT	
-EXPLIST	-EXP	Generate expanded source listing
-NO_EXPLIST *	-NEXP	
-EXTENDED_CHARACTER_SET	-ECS	Change values returned by LOW and COLLATE
-NO_EXTENDED_CHARACTER_SET *	-NECS	
-FRN		Round floating-point numbers in storage
-NO_FRN *	-NFRN	
-FULL_HELP	-FH	Display usage information, option list, and description
-FULL_OPTIMIZE	-FOPT	Optimize fully
-HELP	-H	Display usage information and option list
-INPUT	-I	Designate source file
-LCASE	-LC	Do not convert lowercase

Table 2-1 (continued)
 Summary of Compiler Options and Abbreviations
 (Defaults are marked with asterisks.)

Option	Abbreviation	Significance
-LISTING	-L	Create source listing
-NO_LISTING *	-NL	
-MAP	-MA	List data and procedure names
-NO_MAP *	-NMA	
-MAPWIDE	-MAPW	Set width of listing map
-MAXERRORS	-MAXE	Specify maximum number of reported errors
-NESTING	-NE	Indicate nesting level
-NO_NESTING *	-NNE	
-OFFSET	-OFF	Show offsets in source list
-NO_OFFSET *	-NOFF	
-OPTIMIZE *	-OPT	Optimize object code
-OVERFLOW	-OVF	Enable integer overflow
-NO_OVERFLOW *	-NOVF	
-PRODUCTION	-PROD	Generate production code
-NO_PRODUCTION *	-NPROD	
-RANGE	-RA	Check subscript ranges
-NO_RANGE *	-NRA	
-SILENT	-SI	Suppress warning messages
-SOURCE	-S	Designate source file
-SPACE		Space over time in optimization
-STATISTICS	-STAT	Print compiler statistics
-NO_STATISTICS *	-NSTAT	
-STORE_OWNER_FIELD *	-SOF	Store module names in program code for debugging
-NO_STORE_OWNER_FIELD	-NSOF	
-STRINGSIZE *	-STRZ	Enable STRINGSIZE condition for entire compilation unit
-NO_STRINGSIZE	-NSTRZ	

Table 2-1 (continued)
 Summary of Compiler Options and Abbreviations
 (Defaults are marked with asterisks.)

Option	Abbreviation	Significance
-TIME *		Time over space in optimization
-UPCASE *	-UP	Convert to uppercase
-XREF	-XR	Generate cross-reference
-NO_XREF *	-NXR	

3

Linking and Executing PL/I

INTRODUCTION

This chapter shows you how to create an Executable Program Format (EPF) using BIND. The basic commands given in this chapter will be sufficient for most of your linking needs. For more advanced linking commands with BIND, see the Programmer's Guide to BIND and EPFs.

HOW TO USE BIND

After you have successfully compiled your program, you are ready to link and execute it using BIND. You can do this in either of the following two ways:

- Directly from the PRIMOS command line
- Interactively, by invoking subcommands of BIND

To run BIND from the PRIMOS command line, type:

```
BIND [EPF-filename] [arguments]
```

Invoking BIND in this way allows you to create your runfile (the executable version of your program) in one PRIMOS command line. When using BIND on PRIMOS command level, you must precede each command with a hyphen (-). EPF-filename is the name of the existing EPF or the name

of the object file (binary file) that you want BIND to create. If EPF-filename is missing, BIND uses the name of the first loaded binary file for the EPF-filename base. Arguments given on this command line correspond to internal BIND commands that are explained in the following sections.

A sample linking session using one PRIMOS command line follows:

```
OK, BIND MYPROG -LO ADD -LO SUB -LI PL.LIB -LI  
[BIND rev 19.4]  
BIND COMPLETE  
OK,
```

The commands -LO (or -LOAD) and -LI (or -LIBRARY) are explained below in the section on BASIC LINKING COMMANDS. BIND saves your runfile in your directory with the default name EPF-filename.RUN. In the example above, BIND has saved your runfile with the name MYPROG.RUN.

You may also use BIND interactively by issuing commands to BIND one command line at a time in response to the colon (:) prompt.

To invoke BIND interactively, type the command:

```
BIND
```

BIND then asks you with a colon prompt to load your files. Each time you press the carriage return, you see this prompt. When you leave BIND, your system prompt appears on the screen.

A sample of an interactive linking session using the same program as the single-step link looks like this:

```
OK, BIND MYPROG  
[BIND rev 19.4]  
: LOAD ADD  
: LOAD SUB  
: LI PL.LIB  
: LI  
BIND COMPLETE  
: FILE  
OK,
```

BIND saves the runfile in your directory as MYPROG.RUN.

BASIC LINKING COMMANDS

You can accomplish most of your loading and linking with the following sequence of commands:

1. Load your program with the LOAD command, starting with the main procedure followed by the subprograms in the order in which they are called. (LOAD may be abbreviated LO.)
2. Load the PL/I library with the command LIBRARY PLLIB. (LIBRARY may be abbreviated LI.)
3. Load the system library with the command LIBRARY.
4. When you receive the BIND COMPLETE message, save the EPF runfile and return to PRIMOS level with the command FILE.

The following commands are useful but not necessary to create an EPF. They can be used at any time during the linking sequence.

- MAP can be used to identify references if you do not receive a BIND COMPLETE message at your terminal. MAP -UNDEFINED, the most useful form of this command, produces a list of unresolved references.
- QUIT returns you to PRIMOS level immediately without saving the current EPF.
- HELP gives on-line help if you have problems during a BIND session.

RUNNING YOUR PROGRAM

Once you have compiled and created an EPF, you are ready to run your program using the RESUME command. (For more information on running programs, see the Prime User's Guide.)

The RESUME command has the following format:

```
RESUME [EPF-filename]
```

Previously, we created a runfile with BIND called MYPROG.RUN. To execute that runfile, type the PRIMOS level command:

```
RESUME MYPROG
```

PRIMOS automatically looks in your directory for MYPROG.RUN and begins execution of the EPF.

PL/I Reference Guide

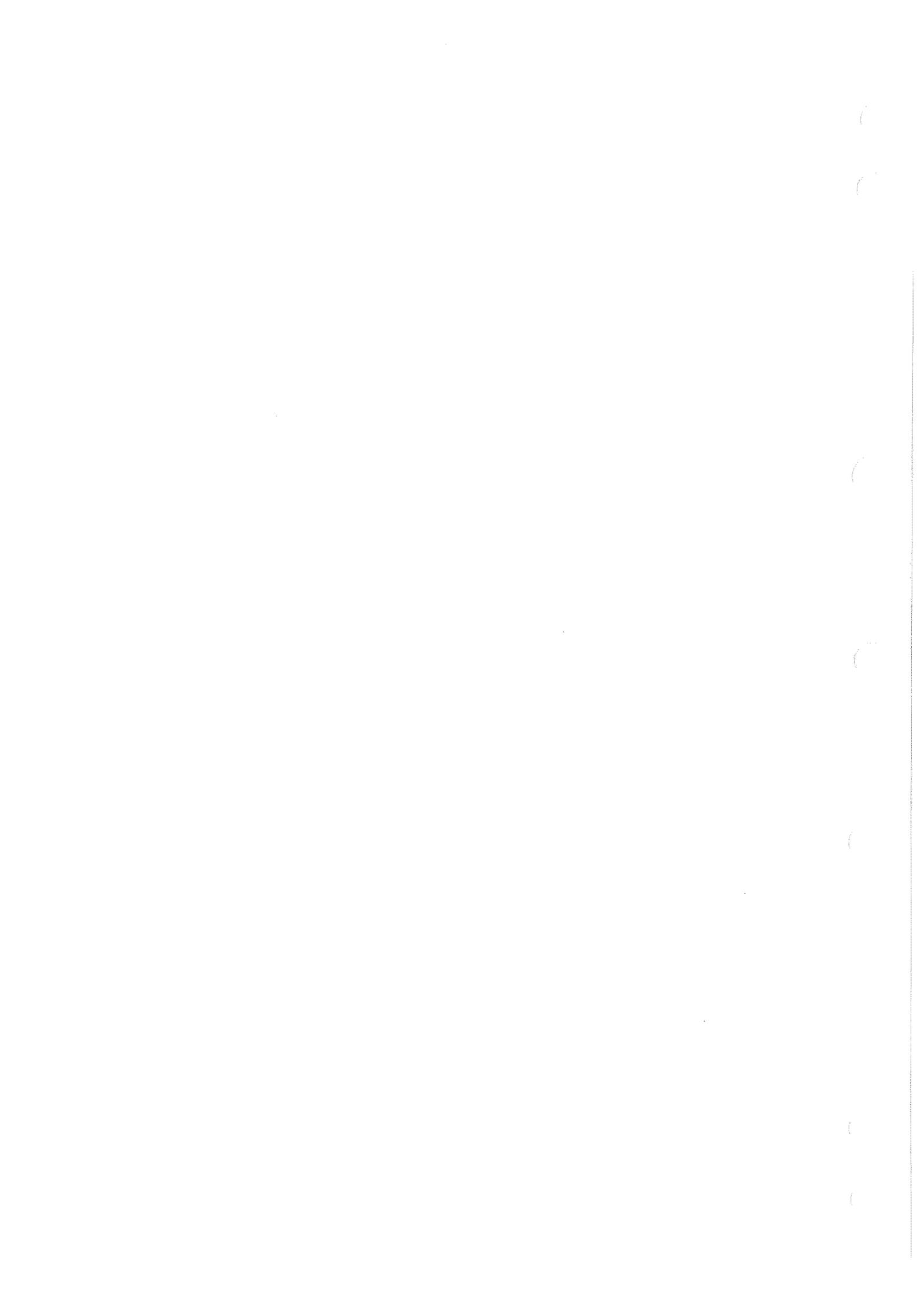
If PRIMOS does not find a file with the name MYPROG.RUN, the following message appears:

Not found. MYPROG (std\$cp)

PRIMOS is telling you that you forgot to create a runfile using BIND.

PART II

Prime PL/I Language Reference



4

The PL/I Language

This section begins by describing enough of the PL/I language to get you started writing simple programs. It assumes that you already know general programming concepts, such as loops, subroutines, functions, input/output, and so forth, and explains how to program these concepts in PL/I.

Much of the material in this section is repeated later in the manual in much more detail. For example, Chapter 9 contains more information on the IF and DO statements. Refer to other chapters of this manual to get further explanations of the topics discussed.

SIMPLE PL/I PROGRAMS

The following simple PL/I program accepts two data values, which it interprets as the two sides of a rectangle. The program then computes the perimeter of the rectangle, using the standard formula, and prints out the result of the computation:

```
TEST:  PROCEDURE OPTIONS(MAIN);  
       GET LIST(X, Y);  
       PERIMETER = 2 * (X + Y);  
       PUT LIST(PERIMETER);  
       END TEST;
```

The following paragraphs describe how this program works.

PL/I Statements

The above program contains five PL/I statements. Each of the five statements is on a separate line, although, as we will explain, this is not required.

PL/I is a free form language. This means that you are not required to begin a statement in any particular column, as you are, for example, in FORTRAN or COBOL. PL/I recognizes the end of a statement by means of a semicolon; each statement must end with a semicolon.

For example, in the above program, the second statement was written as

```
GET LIST(X, Y);
```

However, PL/I rules would have permitted you to write the statement as

```
GET
  LIST
(X, Y);
```

It does not matter that the statement now takes three lines.

Conversely, it is perfectly okay to have several statements on a single line. For example, the second and third statements in the example program above could have been written as

```
GET LIST(X, Y); PERIMETER = 2 * (X + Y);
```

PL/I knows that there are two statements because each of the statements ends in a semicolon.

The maximum size of a source line is 255 characters. A statement may have a maximum of 6143 elements or tokens.

Statement Types

Every PL/I statement has a statement type. Most statements are keyword statements, because the statement type is determined by a special word, called a keyword, in the statement. In the sample program above, all but the third statement are keyword statements. The first two statements are PROCEDURE and GET statements, respectively, and the last two statements are PUT and END statements, respectively. The third statement is an assignment statement and is not determined by a keyword.

The PROCEDURE and END Statements

The sample PL/I program above begins with the following PROCEDURE statement:

```
TEST: PROCEDURE OPTIONS(MAIN);
```

and ends with the following END statement:

```
END TEST;
```

In fact, each PL/I program must begin with a PROCEDURE statement and end with an END statement.

The PROCEDURE statement that begins your program has the following syntax:

```
name: PROCEDURE OPTIONS(MAIN);
```

The name, which should be no more than eight characters long, is the name that you have chosen for your program. The first character must be a letter of the alphabet, and the other seven characters may be either letters or digits. In the sample program above, the name of the program is TEST.

The phrase OPTIONS(MAIN) tells PL/I that this is a main procedure or main program. This phrase is necessary because you can also use the PROCEDURE statement to define subroutines, which are not main programs.

Since PROCEDURE is a fairly long keyword, PL/I permits you to abbreviate it with the alternate keyword PROC. Therefore, the first statement of the sample program above could be written

```
TEST: PROC OPTIONS(MAIN);
```

The use of the abbreviation PROC is entirely equivalent to the use of the full keyword PROCEDURE.

PL/I Reference Guide

The syntax of the END statement that ends the program must be either

```
END;
```

or

```
END name;
```

In the second format, the name is the name of your program, which you have already specified in the PROCEDURE statement. You may use either format. The second format is often preferable because it is more explicit. That is, your program can contain a number of END statements. By specifying the name of the program in the END statement, you are emphasizing the fact that this END statement is the end of the program.

The GET Statement

The second statement of the sample program above is the following GET statement:

```
GET LIST(X, Y);
```

This statement accepts input from your terminal. When PL/I reaches this statement, execution of your program stops, and PL/I waits for you to type values for the variables X and Y. For example, you might type

```
12, 3, CR
```

where CR indicates pressing the return key. PL/I would set the value of X to 12 and the value of Y to 3.

The syntax of the GET statement is

```
GET LIST(variable);
```

or

```
GET LIST(variable, variable, ...);
```

When PL/I executes a statement in one of these formats, it waits for you to type values of the variables appearing in the statement at your terminal.

More complicated formats for the GET statement enable PL/I to

- Accept input from arbitrary files or devices
- Accept formatted input

These are discussed in Chapter 11.

The Assignment Statement

The third statement of the sample program above is an assignment statement:

```
PERIMETER = 2 * (X + Y);
```

PL/I computes the value of $2 * (X + Y)$, and assigns the result to the variable PERIMETER. Since the asterisk is the PL/I symbol for multiplication, PL/I computes $2 * (X + Y)$ by doubling the sum of X and Y.

The simplest syntax of the assignment statement is

```
variable = expression;
```

PL/I computes the value of the expression on the right-hand side of the assignment statement, and assigns that value to the variable on the left-hand side of the statement. A more complex syntax is the following:

```
variable, variable, ... = expression;
```

In this case, PL/I evaluates the expression and assigns the result to each of the variables on the left-hand side of the statement.

The format of an expression is discussed in the section on expressions below and in full detail in Chapter 6.

The PUT Statement

The fourth statement of the sample program is the following:

```
PUT LIST(PERIMETER);
```

Use the PUT statement to print a value on your terminal. The above statement prints the value of the variable PERIMETER on your terminal.

The simplest forms of the PUT statement are

```
PUT LIST(expression);
```

or

```
PUT LIST(expression, expression, ...);
```

PL/I executes such a statement by evaluating each expression and printing its value on your terminal.

More complex forms of the PUT statement perform formatted output or perform output to arbitrary files and devices. See Chapter 11 for details.

You may use any expression in the PUT LIST statement. Therefore, in the sample program at the beginning of this section, you could have replaced the two statements

```
PERIMETER = 2 * (X + Y);  
PUT LIST(PERIMETER);
```

with the single statement

```
PUT LIST(2 * (X + Y));
```

and gotten the same results.

Sometimes you wish to print out some words or text along with your answer. You can enclose such text in apostrophes. For example, consider the following statement:

```
PUT LIST('THE ANSWER IS', 2 * (X + Y));
```

When PL/I executes this statement, if $X = 5$ and $Y = 15$, this PUT statement would print

```
THE ANSWER IS      40
```

The element

```
'THE ANSWER IS'
```

which appears in the PUT statement is a CHARACTER string constant. These are described in the section on the CHARACTER data type below, and in complete detail in Chapter 5.

ELEMENTS OF A PL/I PROGRAM

The following are some of the basic syntactic elements of a PL/I program: identifiers, constants, operators, parentheses, spacing, expressions, and comments.

Identifiers

A PL/I identifier is a sequence of characters that serves as a name. Usually it is the name of the program, a variable, or a keyword. The following is a list of all the identifiers appearing in the sample program given at the beginning of this section:

- The name of the program is TEST.
- The keywords are PROCEDURE, OPTIONS, MAIN, GET, LIST, PUT, END.
- The variables are X, Y, PERIMETER.

Identifiers are used for other purposes besides the three just listed, and these are described in appropriate places in the manual.

You may use any legal identifier as a variable name. A legal identifier must follow these rules:

- The first character of the identifier must be a letter.
- Subsequent characters may be letters, digits, the break character (), also called the underscore, the space character (#), or the dollar sign (\$).
- No more than 32 characters are permitted.

PL/I Reference Guide

The following are legal PL/I identifiers:

```
A  
WHITE  
WHITE68  
RED#40  
YOUR$  
MILES_PER_GALLON
```

On the other hand, the following are not legal identifiers:

```
5XYZ (Does not begin with a letter)  
THIS_IS_A_VERY_LONG_IDENTIFIER_WORD (Too long)
```

If you use a lowercase letter in an identifier, PL/I treats it as if it were an uppercase letter. Therefore, the following would all be considered equivalent variable names:

```
WHITE  
white  
White
```

However, this rule does not apply if you run the PL/I program with the `-LCASE` option explained in Chapter 2. If you specify this option, the three identifiers given just above would be considered three different variables. Note, however, that even if you use the `-LCASE` compiler option, PL/I would still consider list, when used as a keyword, to be the same as LIST.

A feature of the PL/I language is that it has no reserved words, identifiers that are illegal as variables because they are reserved for use as keywords. Therefore, for example, the statement

```
PUT LIST(PUT + LIST);
```

is a perfectly legal PL/I statement, and PL/I recognizes that the first uses of the identifiers `PUT` and `LIST` are as keywords, and that the second uses are as variables. However, as a practical matter, you should avoid using variables that are the same as keywords, because your program will be confusing and difficult to understand.

Constants

A constant is usually a number. The sample program given at the beginning of this section contains only one constant. In the statement

```
PERIMETER = 2 * (X + Y);
```

2 is a constant.

PL/I constants come in many forms, as described in Chapter 5. Numeric constants always begin with a digit or a decimal point. The following are examples of numeric constants:

```
23.4  
89E+12  
1.10F-4BI  
.862
```

PL/I also has string constants, which are described in Chapter 5. Each such constant is enclosed in apostrophes. Some examples of string constants are

```
'THE ANSWER IS'  
'101101'B  
'23BFF43'B4
```

Operators and Parentheses

The assignment statement

```
PERIMETER = 2 * (X + Y);
```

contains two arithmetic operators. These are the asterisk (*) for multiplication and the plus sign (+) for addition. The statement also contains parentheses, and the equal sign for assignment. The use of operators and parentheses in expressions is described briefly in the section on expressions below and in complete detail in Chapter 6.

Spacing

As has been previously stated, PL/I recognizes different individual statements in your program by the fact that each PL/I statement ends with a semicolon. You may write one statement per line if you wish, or you may spread a single statement over several lines, or you may have several statements on a single line.

You may space the statement in any way you wish, subject to some fairly obvious rules. Consider the statement

```
PUT LIST(PERIMETER);
```

You may not insert spaces in the middle of a constant or identifier, or in the middle of a two-character operator, such as `<=`, but you may insert spaces anywhere else you wish. Therefore, the above statement could have been written

```
PUT LIST ( PERIMETER );
```

You must have spaces to separate two identifiers, or two constants, or a constant and an identifier. Therefore, in the above statement, you must have spaces between PUT and LIST, but spaces are optional between LIST and the left parenthesis, and between PERIMETER and the right parenthesis.

Comments

A comment has the syntax

```
/* comment */
```

Any characters between the delimiters `/*` and `*/` are read as a comment. You may insert a comment into your program anywhere you would use a space. For example, you could change the PUT statement shown above to the following:

```
/*FINAL OUTPUT*/PUT LIST(PERIMETER/*ANSWER*/);
```

PL/I considers a comment to be entirely equivalent to a blank.

EXPRESSIONS

Almost every PL/I statement contains expressions. You have already seen how they are used in assignment statements and PUT statements. The full details about expressions are given in Chapter 6. This section deals with some general concepts.

Arithmetic Operators

The following are the PL/I arithmetic operators:

<u>Operator</u>	<u>Meaning</u>
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

You may combine these operators in expressions in any way you wish. For example, if you wish to perform the algebraic assignment

$$\text{let } x = \frac{a + by}{(a - b)^n}$$

use the following assignment statement:

```
X = (A + B * Y) / (A - B) ** N;
```

Priority of Operators

Unless you specify otherwise, by means of parentheses, PL/I performs multiplication and division before addition or subtraction, and exponentiation before any of these. Therefore, the statement

```
X = A * B + C;
```

is equivalent to the statement

```
X = (A * B) + C;
```

PL/I Reference Guide

Furthermore, the statement

$$X = A / B ** C;$$

is equivalent to the statement

$$X = A / (B ** C);$$

In each case, the implied parentheses give higher priority to multiplication over addition, and exponentiation over division.

When you have two adjacent multiplication or division operators, PL/I computes them from left to right. For example, consider the following statement:

$$X = A / B * C;$$

The expression on the right-hand side of the assignment statement contains the operators for division and multiplication. Since there are no parentheses, PL/I computes these from left to right. The result is that the above statement is equivalent to the statement

$$X = (A / B) * C;$$

The same left to right rule applies to adjacent addition and subtraction operators. For example, consider the statement

$$X = A - B + C + D;$$

The addition and subtraction operations are performed left to right, and so this statement is equivalent to

$$X = ((A - B) + C) + D;$$

Note that the operator "-" has two separate meanings. The examples that you have seen for this operator are for subtraction. But the same symbol is also used for a different operation called the negation operation. The negation operation is illustrated by the following statements:

```
X = -A;
Y = -A * B;
```

When used for the negation operation, the "-" operator is called a unary minus and is performed before multiplication or division. Thus the last statement above is equivalent to the statement

```
Y = (-A) * B;
```

PL/I has a number of additional rules for determining the priority of operations in expressions. (See Chapter 6.) The most important rule, however, is

When in doubt, parenthesize!

That is, use parentheses in your expressions whenever you are unsure of the precise priority rules specified by the PL/I language.

FLOW OF CONTROL WITH IF, DO, AND GO TO

Normally, PL/I executes the statements of your program in the order in which they appear. That is, PL/I executes one statement, then the statement following it, and then the statement following that. However, you may use the IF, DO, and GO TO statements to modify the order of execution of the statements of your program.

Conditional Execution With the IF Statement

Consider the following program, which inputs two data values and prints the larger of the two input values:

```
LARGER:  PROC OPTIONS(MAIN);
         GET LIST(FIRST, SECOND);
         IF FIRST > SECOND THEN HIGH = FIRST;
           ELSE HIGH = SECOND;
         PUT LIST(HIGH);
         END LARGER;
```

This program contains a new kind of statement, the IF statement. The particular IF statement in this program says the following: if the value of the variable FIRST is greater than the value of the variable SECOND, then set the variable HIGH equal to FIRST; otherwise, set HIGH to the value of SECOND. Therefore, HIGH is set to equal the larger of the two input values. The PUT statement then prints out the larger value. The simplest syntax of the IF statement is as follows:

```
IF logical-expression THEN statement;  
   ELSE statement;
```

The logical expression is usually a simple comparison (such as FIRST > SECOND in the example above), but may be more complicated, as described below.

The syntax shown above uses two additional statements with the IF statement. The statement following the keyword THEN in the above syntax is called the THEN clause of the IF statement. The one following the keyword ELSE is called the ELSE clause.

PL/I executes the IF statement by evaluating the logical expression to determine whether it is true or false. If it is true, PL/I executes the THEN clause. If the logical expression is false, PL/I executes the ELSE clause.

You may, if you wish, omit the ELSE clause of the IF statement, using the following syntax:

```
IF logical-expression THEN statement;
```

In this case, PL/I evaluates the logical expression, as before. If it is true, PL/I executes the THEN clause. If it is false, PL/I continues sequential execution with the next statement.

Using a DO Group for the THEN or ELSE Clause of the IF Statement

Consider the following program segment:

```
IF X > 0 THEN DO;  
   FLAG = 1;  
   PUT LIST('POSITIVE');  
   END;  
ELSE DO;  
   FLAG = 0;  
   PUT LIST('NEGATIVE OR ZERO');  
   END;
```

This example illustrates how your THEN clause or ELSE clause can include more than just a single PL/I statement. In this example, if the variable X is greater than 0, PL/I sets FLAG to 1 and prints the word POSITIVE; otherwise, PL/I sets FLAG to 0 and prints the words NEGATIVE OR ZERO.

In fact, your THEN or ELSE clause may contain as many statements as you wish. The demarcation structure you use is a DO/END group, a group of statements beginning with a DO statement and ending with an END statement. Other uses of DO/END groups are discussed later in this chapter.

Logical Expressions

A logical expression is a special kind of PL/I expression that is either true or false. For example, in the IF statement illustrated in the sample program above, FIRST > SECOND is a logical expression that is true if the value of the variable FIRST is larger than the value of SECOND, and is false otherwise. In the next example above, X > 0 is true if X is positive, and false if X is 0 or negative.

The simplest form of a logical expression is a comparison. The logical expression FIRST > SECOND is a comparison, using the comparison operator >, which stands for greater than. The following table lists all PL/I comparison operators:

<u>Comparison Operator</u>	<u>Meaning</u>
>	Greater than
<	Less than
=	Equal
<=	Less than or equal
>=	Greater than or equal
^=	Not equal
^>	Not greater than (same as <=)
^<	Not less than (same as >=)

Note

In the Prime EDITOR, the operator NOT must be entered as ^^.

PL/I Reference Guide

For example, the statement

```
IF X + Y <= R * S THEN ...;  
  ELSE ...;
```

is executed as follows: if X plus Y is less than or equal to R * S, PL/I executes the THEN clause; otherwise, PL/I executes the ELSE clause.

A logical expression may also combine comparisons with the logical operators & for AND and | or ! for OR. Consider, for example, this statement:

```
IF (3 < A & A < 5) | A = 0 THEN X = 15;
```

The statement tests whether A is either between 3 and 5 or equal to 0. If the result of this test is true, PL/I sets the value of X to 15.

A word of warning: you may wish to test whether the value of X is between 0 and 10 by using a comparison like the following:

```
IF 0 < X < 10 THEN ...;
```

The problem with this statement is that although it is a legal PL/I statement, it does something quite different from what you expect. (In fact, the result of evaluating $0 < X < 10$ is always true. The full rules for PL/I evaluation of expressions like this are in Chapter 5.) Therefore, PL/I does not tell you that this is an illegal statement, since it is in fact legal. The correct way to test whether X is between 0 and 10 is to use the following:

```
IF 0 < X & X < 10 THEN ...;
```

For introductory purposes, we have been using the words logical expression somewhat imprecisely in this section. Chapter 5 defines a logical expression as a PL/I expression that has the BIT data type. In fact, since PL/I can convert any numeric value to the BIT data type, you may theoretically use any numeric expression as a logical expression. However, most programmers will find these conversion rules somewhat arcane, and so you should restrict your logical expressions to simple comparisons and combinations of comparisons using & and |.

The DO/END Group

In the discussion of the IF statement above, you saw how to use a DO/END group to permit the THEN or ELSE clause to contain more than one statement. One of the fundamental purposes of the DO/END group is to allow you to define a group of statements to be used as a single unit.

By using a different form of the DO statement, you can use PL/I's looping capability. This capability is similar to that provided by the DO statement in FORTRAN, the FOR statement in BASIC, or the PERFORM statement in COBOL. The looping capability is discussed in the next few sections, and in full detail in Chapter 10.

Each DO statement, of whatever format, begins a group of statements to be treated as a unit. For each DO statement in your program there must be a matching END statement that terminates the DO/END group. If the DO statement has no options (and this is the only type of DO statement we have seen so far), the DO/END group is noniterative, meaning that the statements in the group are executed precisely once. If the DO statement has options (which are described below), the group is iterative, since the options control the number of times the group executes, or, in PL/I terminology, the number of iterations of the loop.

The DO WHILE Statement

The following is a program that inputs one or more data values, stopping when the value 0 is reached, and prints out their sum:

```
SUMER:  PROC OPTIONS(MAIN);
        SUM = 0;
        GET LIST(X);
        DO WHILE(X ^= 0);
            SUM = SUM + X;
            GET LIST(X);
        END;
        PUT LIST(SUM);
        END SUMER;
```

This program contains a DO/END group that begins with the following statement:

```
DO WHILE(X ^= 0);
```

This statement says, execute the statement inside the DO/END group over and over, as long as X does not equal 0.

The syntax for a group of this type is as follows:

```
DO WHILE(logical-expression);  
...  
...  
END;
```

The logical expression has the same form in the DO WHILE statement as it does for the IF statement described before. PL/I executes all the statements in the group over and over, as long as the logical expression is true.

Be aware that so-called zero-trip DO loops are possible in PL/I. If the logical expression is false the first time that it is tested, PL/I does not execute the statements inside the group at all; execution continues immediately with the statement following the END statement. In the SUMER example shown above, if the first input value for X is 0, the statements inside the group does not execute, and execution continues immediately with the PUT statement following the END statement, which prints the value 0 for SUM.

DO/END Groups with Index Variables

Users of other programming languages may be more comfortable with the form of the DO statement illustrated in the following program segment:

```
DO K = 5 TO 20;  
PUT LIST(K);  
END;
```

This program segment prints out the integers 5, 6, 7, ..., 20. The DO statement says, execute all the statements in the DO group 16 times. The first time, let K equal 5; the second time, let K equal 6; and so forth, until the sixteenth time, when K equals 20.

The variable K is called the index variable of the DO/END group. For each iteration of the statements inside the group, PL/I first changes the value of the index variable in the manner dictated by the options of the DO statement. The options of the DO statement also determine the number of iterations. In the example above, the DO statement says that K is to have the value 5 for the first iteration, that the value of K is to be increased by one for each subsequent iteration, and that the loop is to terminate when the value of K exceeds 20.

There are many forms of DO statements with an index variable. See Chapter 10 for details. This introductory section shows some examples illustrating the most useful options of the DO statement with an index.

In the example above, PL/I increments K by 1 for each repetition of the loop. If you wish to increment by a different value, use the BY option. For example, the loop

```
DO K = 5 TO 20 BY 3;
  PUT LIST(K);
END;
```

prints the values 5, 8, 11, 14, 17, 20. The TO and BY clauses may go in either order. Therefore, the loop

```
DO K = 5 BY 3 TO 20;
  PUT LIST(K);
END;
```

is equivalent.

Any of the three values shown in the DO statement in the last example may be an arbitrary PL/I expression. For example, a DO group like

```
DO COUNT = X + 3 TO 4 BY U + Z;
  ...
END;
```

is legal.

A final useful form permits you to make a list of the values that you wish the index variable to take. For example, the program segment

```
DO VALUE = 8, 12, -3, 4, 15, 2;
  PUT LIST(VALUE);
END;
```

prints the values 8, 12, -3, 4, 15, and 2. Each of the specifications in the list may contain a TO and BY option. For example, the loop

```
DO VALUE = 3 TO 5, 8 TO 10, 15 TO 21 BY 2;
  PUT LIST(VALUE);
END;
```

prints 3, 4, 5, 8, 9, 10, 15, 17, 19, and 21.

There are further examples of these DO statements later in this section, especially in the discussion of arrays. For complete details, see Chapter 10.

Named DO/END Groups

Recall that each PL/I program begins with a PROCEDURE statement specifying the name of your program and ends with an END statement specifying the same name. The syntax is

```
name: PROC OPTIONS(MAIN);  
    ...  
    ...  
    END name;
```

The END statement matches the PROCEDURE statement. The name specified in the END statement emphasizes that fact, since it is the same name that appears in the PROCEDURE statement.

Just as you use an END statement to match a PROCEDURE statement, you must also use it to match a DO statement. There may be many DO statements in your program. Each must have a corresponding END statement. It is even possible to have one DO/END group nested inside another group.

In order to make your program easier to understand, you may wish to name your DO/END groups in the same way that you name your entire program. For example, the SUMER program illustrated in the section on The DO WHILE Statement could be changed as follows:

```
SUMER:  PROCEDURE OPTIONS(MAIN);  
        SUM = 0;  
        GET LIST(X);  
INLOOP: DO WHILE(X ^= 0);  
        SUM = SUM + X;  
        GET LIST(X);  
        END INLOOP;  
        PUT LIST(SUM);  
        END SUMER;
```

In this revised example, the DO statement has a statement name of INLOOP. This same name is referenced in the corresponding END statement.

Multiple Closure END Statements

In certain circumstances, you may use a single END statement to terminate several DO statements simultaneously. A statement of the form

```
END name;
```

closes off all unclosed nested DO groups back to the one whose statement name is given in the END statement. For example, consider the following:

```
LOOP1: DO ...;
        ...
        DO ...;
        ...
        DO ...;
        ...
        END LOOP1;
```

The END statement in this example closes off all three nested DO groups.

Statement Labels and the GO TO Statement

You have already seen how you can name a program or a DO/END group by putting a statement name on the PROCEDURE or DO statement, respectively. The syntax is

```
name: PROC OPTIONS(MAIN);
```

for the PROCEDURE statement, and

```
name: DO options;
```

for the DO statement. In either case, the name is an identifier of your choice for the name of your program or of the group, respectively.

You may use a similar syntax to provide a statement name, or statement label, for any executable statement, and then you may use the GO TO statement to cause your program to transfer control to the labelled statement. The syntax of the GO TO statement is either of the following:

```
GO TO name;
```

or

```
GOTO name;
```

where the name is the statement label of a statement.

You have previously studied the SUMER program, which inputs data values until the value zero is read as input, and then prints the sum of the input values. The original program of this name was written using DO WHILE. To write the same program using GO TO, type

```
SUMER:   PROC OPTIONS(MAIN);
         SUM = 0;
LOOP:    GET LIST(X);
         IF X = 0 THEN GO TO PUT_STMP;
         SUM = SUM + X;
         GO TO LOOP;
PUT_STMP: PUT LIST(SUM);
         END SUMER;
```

Although this program works the same as the program using DO WHILE, most practitioners of structured-programming techniques find the DO WHILE form of the program to be far clearer and more maintainable than the above version using the GO TO statement.

NUMERIC DATA TYPES

In all the examples so far in this section, the constants and variables always had integer values. If you wish to use noninteger data, you must take some special precautions. For full details, see Chapter 5.

DECLARE Statement for FIXED Integer Data Types

In the discussion of DO WHILE, a program called SUMER was used as an example. Consider the following modification of that program example:

```
SUMER:  PROC OPTIONS(MAIN);
        DECLARE X FIXED DECIMAL(3);
        DECLARE SUM FIXED DECIMAL(5);
        SUM = 0;
        GET LIST(X);
        DO WHILE(X ^= 0);
            SUM = SUM + X;
            GET LIST(X);
        END;
        PUT LIST(SUM);
        END SUMER;
```

This program contains two DECLARE statements, which define the respective data types of the variables X and SUM; that is, they specify the type of data that these variables can accommodate.

The first of the DECLARE statements is as follows:

```
DECLARE X FIXED DECIMAL(3);
```

This statement specifies that the variable X is to have certain data type attributes, FIXED DECIMAL(3), which have the following significance:

- The variable X is given the FIXED attribute. This means that the scale of the data type of X is FIXED. (The other possible scale data type attribute is FLOAT; this is discussed later.) For now, a scale of FIXED means that X can only have integer values, but this is modified in the section Noninteger FIXED Data Types.
- The variable X is given the DECIMAL attribute. This means that the base of the data type of X is DECIMAL. (The other possible base data type is BINARY; this is discussed in Chapter 5.) A base of DECIMAL means that PL/I uses internal data representation for X with decimal digits.
- The element 3, following DECIMAL in the declaration, specifies that the precision or number of digits in the data type of X is 3.

Putting all the above together, you see that X is given the data type FIXED DECIMAL(3), which means the following: X can have an integer value containing three decimal digits. Such values may be either positive or negative. Therefore, X can have any integer value greater than or equal to -999 and less than or equal to +999. Typical values that X can have are as follows:

-876	+000
-423	+005
-029	+042
-005	+259

Notice that, in the above list of values that X can have, each number is shown with all three digits and the sign, in order to emphasize that the data type is FIXED DECIMAL(3). Similar representations are used elsewhere in this guide to emphasize the data type of the numeric value.

In the current case, the variable X may have any integer value containing three decimal digits. When your program assigns a different kind of value to X, as in the statement

```
X = 28.9;
```

then PL/I truncates the value being assigned, and throws away the fractional part. The result is that X is assigned the value +028. On the other hand, it is illegal to assign to X a value such as 1842, which is larger than +999.

In the actual program shown at the beginning of this section, the variable X is assigned a value by means of a GET LIST statement, rather than by means of an assignment statement. However, the rules are the same. When PL/I reaches the GET LIST statement, execution stops and PL/I waits for you to type the input value of X on your terminal. If your typed input value is fractional, PL/I truncates it to an integer. For example, if you type

```
28.9
```

then PL/I would assign to X the value +028. An input value larger than +999 or smaller than -999 would be illegal.

The variable SUM in the sample program is declared with the following statement:

```
DECLARE SUM FIXED DECIMAL(5);
```

Therefore, the variable SUM has the attributes FIXED DECIMAL(5), which differ from the data type attributes for X only in that the precision or number of digits is 5 instead of 3. The result is that the variable SUM can have any integer value between -99999 and +99999, inclusive.

In view of these declarations, the above program operates properly only if the input values are between -999 and +999, and if the sum never exceeds five digits. Any noninteger input values are truncated.

Default Data Types

Every variable of your program has a data type, whether you use a DECLARE statement for the variable or not. If you do not declare the variable, PL/I gives it the default attributes FIXED BINARY (15). The base of this data type is BINARY (rather than decimal), and so a variable with default attributes can accommodate any integer value with up to 15 binary digits (or bits). Therefore, such a variable can have any value between -32768 and +32767, inclusive.

Noninteger FIXED Data Types

Up to now, it has been implied that a variable whose data type is FIXED can only take on integer data values. That statement was not entirely true. Consider, for example, the following declaration:

```
DECLARE SALARY FIXED DECIMAL(7, 2);
```

For this declaration, the data type of the variable SALARY has a scale of FIXED, a base of DECIMAL, and a precision of (7, 2). The precision contains two integers, a number-of-digits value of 7, and a scale-factor value of 2. This means that SALARY can take on any value that can be represented in seven decimal digits, with two digits following the decimal point. Therefore, SALARY can have, for example, any of the following values:

```
+87429.78      +00000.00
+00723.00      -00045.16
+00005.10      -97423.90
```

In fact, SALARY can have any value in this format from -99999.99 to +99999.99.

Similarly, you can declare a variable to have other precisions and scale factors. For example, if you declare

```
DECLARE VR FIXED DECIMAL(10, 3);
```

then the variable VR can have any value from -9999999.999 to +9999999.999 that can be represented in ten decimal digits, with three digits following the decimal point. The maximum precision is fourteen digits.

FLOAT Data Types

If the data type of a numerical variable has a scale of FIXED, the decimal point is always in a fixed position with respect to the digits in the value. For example, the variable SALARY described above can have a value with seven decimal digits, and the decimal point is always in a fixed position, two digits from the end.

If the data type of a numeric variable has a scale of FLOAT rather than FIXED, the position of the decimal point floats with respect to the digit in the variable. For example, consider the following declaration:

```
DECLARE RANGE FLOAT DECIMAL(5);
```

The value of the variable RANGE can have up to five significant digits, and the decimal point can be in any position with respect to those digits. Therefore, RANGE can have such values as +8.7942, +1142.3, or -.11015.

In this manual, to emphasize that a value is a FLOAT value, that value is written in a special notational form corresponding to scientific notation. The idea behind this format is that a FLOAT value really has two parts: the significant digits in the value, and the position of the decimal point with respect to those digits.

For example, when we write

```
2.84E5
```

we are representing the value 2.84×10^5 , which equals 284000. The format 2.84E5 emphasizes the fact that the value has three significant digits. The value could also be written

```
284E3
```

with the same effect. The value to the right of the letter E specifies the number of places to move the decimal point to the right. If the value following the letter E is negative, the decimal point is moved to the left. Therefore, the number

284E-3

has the same value as .284.

Returning to the declaration of the FLOAT variable RANGE shown above, RANGE may have any FLOAT value containing five significant digits. Thus, for example, RANGE can have any of the following values:

```
+8.9742E0
+4.2675E20
-4.7426E-15
-.87200E-2
+00000E0
```

The maximum precision is fourteen digits.

Conversions Among Numeric Variables

PL/I supports many data types, and it follows the general rule that when you use a variable or expression of one data type in an environment that requires a different data type, PL/I converts the value of the variable or expression to the correct data type.

The example of this concept that is easiest to understand is in the assignment statement. Consider the following program segment:

```
DECLARE X FLOAT DECIMAL(8);
DECLARE K FIXED DECIMAL(5);
...
K = X;
...
X = K;
```

The first assignment statement, $K = X$, assigns a FLOAT value to a FIXED variable. PL/I converts the FLOAT value to FIXED, truncating a noninteger value, if necessary. The second assignment statement, $X = K$, assigns a FIXED value to a FLOAT variable. Here the conversion does not require truncation or other change of value, but it does require a change to the internal representation of the value from a fixed-point representation to a floating-point representation.

Another more complicated case is a statement like the following:

$$K = X + K;$$

where you assume that the declarations are the same as above. Because PL/I cannot directly compute the sum of a FLOAT and FIXED value, it converts the value of K from FIXED to FLOAT, storing the result of the conversion in a temporary location. PL/I then adds the two FLOAT values to get a FLOAT sum, converts this result to FIXED, and assigns the result of this conversion to the variable K.

These examples illustrate the following rules:

- If an expression involves both FIXED and FLOAT values, PL/I converts the FIXED value to FLOAT in order to compute the value of the expression.
- If the right-hand side of an assignment statement has a different data type from the variable on the left-hand side, PL/I converts the value of the right-hand side to the data type of the variable on the left-hand side before doing the assignment.

These two rules are simplifications of a collection of fairly complicated rules involving expression evaluation and conversion. See Chapter 6 for the full set of these rules.

BUILT-IN FUNCTIONS

PL/I's built-in functions give you additional capabilities not provided by the ordinary operators (+, -, *, etc.). For example, the statement

$$A = B + C;$$

uses the + operator to compute the sum of the values of B and C, and stores the results in A. But there is no PL/I operator to compute the maximum of two values. However, you can use the built-in function MAX as follows:

$$A = \text{MAX}(B, C);$$

This statement computes the maximum of B and C and stores the result in A.

When discussing PL/I built-in functions, some special terminology is used:

- The arguments of the built-in functions are the values enclosed in parentheses following the name of the built-in function. For example, in the last assignment statement, the arguments of the MAX built-in function are B and C.
- A reference to a built-in function is the use of that built-in function in a statement. For example, in the last assignment statement, MAX(B, C) is a reference to the MAX built-in function.
- The word returns is used to describe the result that PL/I computes for the function. For example, you could say that the MAX built-in function returns a value equal to the maximum of the values of its arguments.

The argument to a built-in function may be any PL/I expression. For example, consider the following statement:

```
A = MAX(B * C, X + Y + 5);
```

This statement is legal. PL/I evaluates the expressions B * C and X + Y - 5 and returns the larger of these two values; the value returned is then assigned to A.

You may use a built-in function within any PL/I expression, in any statement. Consider, for example, the following:

```
A = 3 + MAX(B * C, X) + MAX(Q, R);
```

This is an assignment statement whose right-hand side is an expression containing two different references to the MAX built-in function. PL/I adds 3 to the sum of the respective values returned by these two references to MAX, and assigns the result of that computation to the variable A.

In some cases, the PL/I built-in functions are simply conveniences. For example,

```
A = MAX(B, C);
```

is a convenient method for assigning to A the maximum of the values of B and C. However, the statements

```
IF B > C THEN A = B;  
ELSE A = C;
```

do the same thing by using an IF statement rather than the MAX built-in function.

In the following sections, some of the most commonly used PL/I built-in functions discussed. For full details on these built-in functions, see Chapter 14.

Arithmetic Built-in Functions

The PL/I arithmetic built-in functions perform simple arithmetic computations.

ABS: The ABS built-in function returns the absolute value of its argument. Consider the following:

```
PUT LIST(ABS(X));
```

This statement prints the absolute value of X. The absolute value of a number is that number with the sign made positive. Therefore, ABS(5) returns 5, and ABS(-5) also returns 5. The above PUT statement could have been replaced with

```
IF X >= 0 THEN PUT LIST(X);  
ELSE PUT LIST(-X);
```

which does exactly the same thing.

TRUNC, CEIL, FLOOR: There are three related built-in functions, TRUNC, CEIL, and FLOOR, which take a (possibly) noninteger argument and return an integer value. These three functions operate as follows on a noninteger argument:

- FLOOR of a noninteger argument returns the next lower integer. For example, FLOOR(2.7) returns the value 2, and FLOOR(-2.7) returns -3.

- CEIL of a noninteger argument returns the next higher integer. For example, CEIL(2.7) returns the value 3, and CEIL(-2.7) returns -2.
- TRUNC of a noninteger argument returns the integer obtained by truncating the argument. For example, TRUNC(2.7) returns the value 2, and TRUNC(-2.7) returns -2.

All three functions leave an integer argument unchanged. For example, FLOOR(5), CEIL(5), and TRUNC(5) all return the value 5.

Notice that FLOOR and TRUNC return the same value for positive arguments, and CEIL and TRUNC return the same values for negative arguments.

MOD: The MOD built-in function takes two arguments and returns the remainder that results when the first argument is divided by the second. For example, MOD(17, 5) returns the value 2, since 17 divided by 5 has a quotient of 3 and a remainder of 2. One use of MOD is to determine whether an integer value is odd or even. For example,

```
IF MOD(K, 2) = 0 THEN PUT LIST('EVEN');
   ELSE PUT LIST('ODD');
```

prints the word EVEN or ODD, depending upon whether the value of K is even or odd.

MAX, MIN: You saw an example of the MAX built-in function above. The two related functions, MAX and MIN, return the maximum and minimum, respectively, of the values of their arguments. Either may have two or more arguments. For example, the statement

```
A = MAX(B, C, D);
```

assigns to A the maximum of the values of B, C, and D. As a further example, consider

```
R = MIN(Q * S, O, S + T, X);
```

which assigns to R the minimum of the four values shown in the arguments.

Mathematical Built-in Functions

These built-in functions are useful in mathematical applications. In most cases, PL/I uses a polynomial approximation to compute the value returned by the function.

The `SQRT` built-in function returns the square root of the argument, the number that, when multiplied by itself, results in the value of the argument. Therefore, `SQRT(25)` returns 5, since $5 * 5 = 25$. `SQRT(2.000)` returns 1.414.

There are two sets of trigonometric built-in functions, one in which the angle is measured in degrees, and one in which the angle is measured in radians.

`SIN(X)` returns the sine of X, where X is measured in radians, and `SIND(X)` returns the sine of X, where X is measured in degrees. For example, `SIND(90)` returns the value 1, while `SIN(3.14159/2)` returns the value 1. Similarly, `COS` and `COSD` compute the cosine of the argument measured in radians and degrees, respectively, and `TAN` and `TAND` compute the tangent.

The reference `EXP(x)` computes e^x where e is the transcendental number approximately equal to 2.71828.

There are three logarithm functions, `LOG`, `LOG10`, and `LOG2`. `LOG` computes the natural logarithm (logarithm to base e) of the argument. `LOG10` computes the common logarithm (logarithm to base 10) of the argument, and `LOG2` computes the logarithm to base 2.

There are a number of other mathematical built-in functions. (See Chapter 14.)

CHARACTER STRING DATA TYPE

Most variables have a numeric data type. That is, the value of the variable is a numeric value. The type of numeric value (for example, whether it must be an integer) depends upon the data type of the variable.

This section discusses a different kind of data type, the `CHARACTER` data type. The value of a `CHARACTER` variable is not a number, but rather a string of characters.

The following paragraphs explain generally how this works. For a full description of the `CHARACTER` data type, see Chapter 5.

The CHARACTER String Declaration

Consider the following PL/I statements:

```

DECLARE C CHARACTER(5);
...
C = 'SMITH';
PUT LIST(C);

```

The first of these statements is a `DECLARE` statement that specifies that the variable `C` is to have the `CHARACTER` data type attribute, and that the value of `C` is to be a string of five characters. The assignment statement in the example assigns to `C` the characters 'SMITH'. The `PUT` statement prints

```
SMITH
```

since these are the five characters in the string value of `C`.

The value of `C` is always a string of precisely five characters, neither more nor less. On the other hand, if you declare `CV` as follows:

```
DECLARE CV CHARACTER(5) VARYING;
```

then the value of the variable `CV` can have five or fewer characters.

If you assign to either `C` or `CV` a string of length greater than five characters, PL/I truncates the string before assigning it. Consider, for example, the following assignments:

```

C = 'JOHNSON';
CV = 'JOHNSON';

```

In each of these cases, the string being assigned is too long for the variable it is being assigned to, and so PL/I assigns to each of `C` and `CV` the truncated string value 'JOHNS'.

The difference between `C` and `CV` is illustrated when you assign a string shorter than five characters. Consider, for example, these statements:

```

C = 'ABC';
CV = 'ABC';

```

These two assignment statements are similar, but the results are different, since C is CHARACTER and CV is CHARACTER VARYING. PL/I assigns to C the value 'ABCbb', where b is a blank character; PL/I pads the string 'ABC' with blanks to get a total padded length of five, the length required for assignment to C. On the other hand, PL/I assigns to CV the string 'ABC'; no padding is done because CV has the VARYING attribute, and can have a length of five or less. Therefore, the length of C is always five, since PL/I pads a short string value with blank characters to a length of five, but the value of CV can have any length of five or less.

CHARACTER String Constants

A string of characters enclosed between apostrophes is called a CHARACTER string constant. Consider, for example, the following two statements:

```
PUT LIST('THE ANSWER IS', X);  
CV = 'JOHNSON';
```

Each of these statements contains a CHARACTER string constant. The first contains 'THE ANSWER IS', and the second contains 'JOHNSON'.

Normally, you can put any characters you want between the apostrophes, so that they will be in the CHARACTER string. However, special problems arise when you wish to put an apostrophe itself into the CHARACTER string constant. If you wish to do this, use two apostrophes. Consider, for example, the following statement:

```
PUT LIST('I DON''T KNOW.');
```

This PUT statement prints

```
I DON'T KNOW.
```

The two apostrophes between the N and the T in the CHARACTER string constant are printed as a single apostrophe.

There is a special CHARACTER string constant called the null string. For example, consider these statements:

```
DECLARE CV CHARACTER(5) VARYING;  
...  
CV = '';
```

The assignment statement assigns to CV the null string, a string containing no characters at all. The result is that CV has a value whose length is 0.

GET LIST With CHARACTER String Variables

You may use GET LIST to input the value of a CHARACTER string variable. For example, if WORD is a CHARACTER string variable, the statement

```
GET LIST(WORD);
```

inputs a value for the variable WORD from your terminal. When execution of your program stops to wait for you to type an input value for the GET statement, type any CHARACTER string constant, followed by a blank or comma, as your input value. For example, if you type

```
TURKEY
```

then that CHARACTER string value is assigned to WORD.

OPERATIONS ON CHARACTER STRINGS

You can perform operations on CHARACTER string data, just as you perform operations on numeric data. The operations on numeric data are usually addition, subtraction, multiplication, and division. For string data, the operations do such things as pulling strings apart and putting them together.

For numeric operations, PL/I uses the commonly accepted symbols +, -, *, /, and ** to represent the operations. Other operations on character strings are described below.

The Concatenation Operator

The CHARACTER string operation for concatenation can be represented either by the symbol || or the symbol !!. For example, consider the following statements:

```
DECLARE C1 CHARACTER(20) VARYING;  
DECLARE C2 CHARACTER(4);  
DECLARE C3 CHARACTER(10) VARYING;  
...  
C1 = 'QRS';  
C2 = 'ABCD';  
C3 = C1 || C2;
```

The last assignment statement uses the concatenation operator. PL/I concatenates the two string values by sticking them together end to end. The result is that C3 is assigned the value 'QRSABCD'.

CHARACTER String Comparisons

Just as you may compare two numbers to determine which is greater, you may also compare two CHARACTER string values. See Chapter 6 for the precise meaning of CHARACTER string comparisons.

For the purposes of this introductory section, CHARACTER string comparisons are somewhat like comparing two words for their relative position in alphabetical order. That is, the word ANT comes before the word ANVIL in a dictionary, and so, PL/I considers the CHARACTER string 'ANT' to be less than the CHARACTER string 'ANVIL'.

The preceding section contained a program segment example illustrating the concatenation operator. The following statements use the same variables:

```
IF C1 < C2 THEN X = 1;  
ELSE X = 2;
```

Recall that C1 = 'QRS' and C2 = 'ABCD'. Since QRS comes after ABCD in alphabetical order, the comparison C1 < C2 is false, so PL/I sets X equal to 2.

The above description of CHARACTER string comparisons is greatly simplified. In fact, CHARACTER strings may contain more than letters; they may also contain digits, blanks, and punctuation symbols, for example. The complete set of characters that you may use is called the ASCII collating sequence. (See Appendix B.) For a full set of rules for comparing strings of characters, see Chapter 6.

CHARACTER String Built-in Functions

Most CHARACTER string operations are performed by means of built-in functions rather than operators. Therefore, to manipulate CHARACTER string values, we must use CHARACTER string built-in functions.

The LENGTH built-in function takes a CHARACTER string argument and then returns the length of the argument. The length of the string is the number of characters in the string.

Several of the built-in functions use the concept of the position of a substring of a string. For example, start with the string 'JOHNSON'. Then 'OHN' is a substring starting in position 2, since it starts at the second character position of 'JOHNSON'. Similarly, 'JOHN' is a substring in position 1. On the other hand, 'JOS' is not a substring of 'JOHNSON', since the characters of 'JOS' do not appear in consecutive positions in 'JOHNSON'.

The built-in function reference

```
SUBSTR(c, m, n)
```

returns the substring of string c starting at position m and going for n characters. For example, the reference

```
SUBSTR('JOHNSON', 4, 3)
```

returns the substring 'NSO'. Similarly, the reference

```
SUBSTR('JOHNSON', 7, 1)
```

returns 'N'.

As an example of the use of these functions, consider the following PL/I program:

```
SPLIT: PROC OPTIONS(MAIN);
        DECLARE STR CHARACTER(100) VARYING;
        GET LIST(STR);
        DO POSITION = 1 TO LENGTH(STR);
            PUT SKIP LIST(SUBSTR(STR, POSITION, 1));
        END;
    END SPLIT;
```

This program inputs a CHARACTER string, and prints out the string, one character per line of output. For example, if the input is 'ABC', the output is

```
A
B
C
```

This program works as follows:

- The DECLARE statement specifies that STR is a CHARACTER string variable whose maximum length is 100.
- The GET statement accepts a CHARACTER string value from your terminal, and stores it into the variable STR.
- The DO statement uses an index variable called POSITION, which varies from 1 to the number of characters in the input string, as determined by use of the built-in function LENGTH. Therefore, there is one execution of the loop for each character in the input string.
- The PUT string prints out a single character on a new line. The SKIP option in the PUT statement specifies that output is to be on a new line. The reference to SUBSTR(STR, POSITION, 1) returns a string containing the single character in the position determined by the variable POSITION.

The net result is that the DO loop prints each character of the input string on a new line of output.

The SUBSTR built-in function may also be used with only two arguments, rather than three. A reference to

```
SUBSTR(c, m)
```

returns the substring of string c starting at position m and going to the end of string c. For example, a reference to

```
SUBSTR('JOHNSON', 5)
```

returns the substring 'SON'.

You may use the INDEX built-in function to perform string searches. The reference

```
INDEX(c, s)
```

returns an integer value equal to the position of the first occurrence of s as a substring of c. If s is not a substring of c, the built-in function reference returns 0. For example, the reference

```
INDEX('JOHNSON', 'HN')
```

returns the integer value 3, since 'HN' is a substring of 'JOHNSON' starting at position 3. The reference

```
INDEX('JOHNSON', 'N')
```

returns the integer value 4, since the first occurrence of 'N' in 'JOHNSON' is at position 4. On the other hand, the reference

```
INDEX('JOHNSON', 'JOS')
```

returns the integer value 0, since 'JOS' is not a substring of 'JOHNSON'.

Here is an example of a program that uses several of these built-in functions:

```
WORDS: PROC OPTIONS(MAIN);
        DECLARE SENTENCE CHARACTER(200) VARYING;
        GET LIST(SENTENCE);
        K = INDEX(SENTENCE, ' ');
        DO WHILE(K > 0);
            PUT SKIP LIST(SUBSTR(SENTENCE, 1, K - 1));
            SENTENCE = SUBSTR(SENTENCE, K + 1);
            K = INDEX(SENTENCE, ' ');
        END;
        PUT SKIP LIST(SENTENCE);
    END WORDS;
```

This program inputs a CHARACTER string value that is interpreted as a sentence. The program then prints the sentence out, one word per line of output. For example, if the input string is

```
THIS IS A STRING.,
```

then the program prints

```
THIS  
IS  
A  
STRING.
```

The program assumes that a single blank separates each adjacent pair of words in the sentence. The program works as follows:

1. The GET statement inputs a CHARACTER string into the string variable SENTENCE.
2. The assignment statement

```
K = INDEX(SENTENCE, ' ');
```

computes the position of the first blank character in the variable SENTENCE, and then assigns the result to the integer variable K. If SENTENCE contains no blank characters, K is set to 0. Notice that this statement appears in two places, just before the DO statement and just before the END statement that terminates the DO group, so that K has a valid value each time the loop is iterated.

3. The DO statement specifies that looping should continue as long as K is positive; that is, looping continues as long as SENTENCE contains a blank character.
4. The following PUT statement appears in the loop:

```
PUT SKIP LIST(SUBSTR(SENTENCE, 1, K - 1));
```

This PUT statement prints the first word in the variable SENTENCE. It does this by printing the substring of SENTENCE containing all characters up to, but not including, the first blank character in SENTENCE. For further explanation, see the examples below.

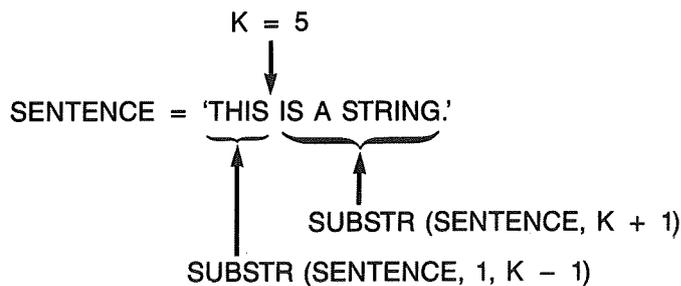
5. The next assignment statement in the loop,

```
SENTENCE = SUBSTR(SENTENCE, K + 1);
```

recomputes the value of the variable `SENTENCE` by removing the first word (and the blank following) from the front of the string value of `SENTENCE`. For further explanation, see the examples below.

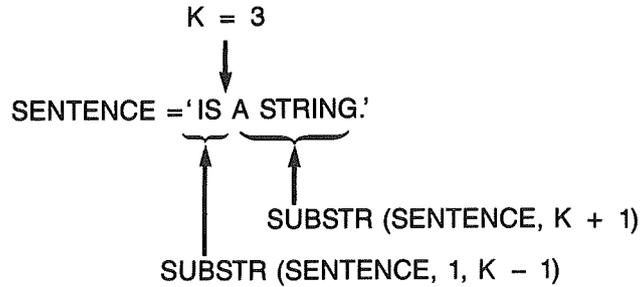
6. The `DO` group terminates when there are no more blanks in `SENTENCE`, which happens when `SENTENCE` contains a single word. At that point, the `PUT` statement just before the end of the program prints the last word in `SENTENCE`.

To understand how this program works, consider the following:

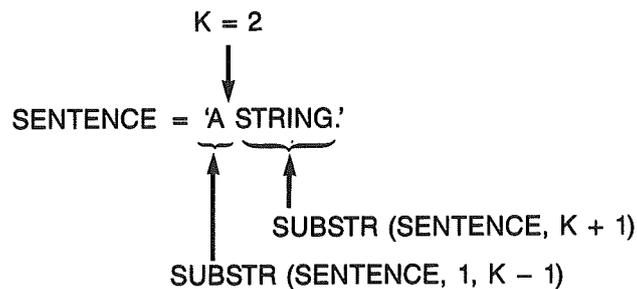


If `SENTENCE` has the value shown above, $K = 5$, since the first blank occurs in the fifth position of `SENTENCE`. The program then prints the value of `SUBSTR(SENTENCE, 1, K - 1)`, which is the first word in the sentence. Then the program recomputes the value of `SENTENCE` to the value of `SUBSTR(SENTENCE, K + 1)`, which throws away the first word and blank of `SENTENCE`.

For the second execution of the loop, you have



After the PUT statement and the recomputation of SENTENCE, you have the following for the third execution:



After the end of the third execution, SENTENCE equals 'STRING.', which contains no blanks; the value of K is 0, and there is no fourth execution of the loop. The final PUT statement prints out 'STRING.'.

For other string built-in functions, see Chapter 14.

ARRAYS AND STRUCTURES

Up to this point, each variable that has been used in the examples of this section has been what is called a scalar, meaning that the variable represents precisely one data value. By using the DECLARE statement, you may specify that a given variable is to represent many data values all by itself. Such a variable is called an aggregate. There are two kinds of PL/I aggregates: arrays and structures.

Arrays

What is called an array in the PL/I programming language is called a dimensioned variable in other programming languages. Consider the declaration

```
DECLARE SLOPES(5) FIXED DECIMAL(9);
```

which specifies that SLOPES is an array of five data values, each of which is an integer that accommodates nine decimal digits. You may refer to the five individual data values by means of the following kinds of references:

```
SLOPES(1)
SLOPES(2)
SLOPES(3)
SLOPES(4)
SLOPES(5)
```

Therefore, for example, you could use statements like

```
SLOPES(3) = 15;
SLOPES(2) = SLOPES(3) + 5;
```

to set or fetch the various individual data values of the array SLOPES. The numbers in parentheses are called subscripts of the array SLOPES. For example, in the last assignment statement, the first reference to SLOPES has a subscript of 2, and the second reference has a subscript of 3.

The main advantage of the use of arrays is that you may use variables or expressions in the subscripts. Consider, for example, the statements

```
DO K = 1 TO 5;
SLOPES(K) = SLOPES(K) - 1;
END;
```

which make up a DO loop that decreases each element of the array SLOPES by one.

You may also use a reference to the array SLOPES with no subscript at all, to refer to all five elements of the array simultaneously. For example, consider the assignment statement

```
SLOPES = 25;
```

which assigns the value 25 to each of the five elements of the array SLOPES. Furthermore, the statement

```
PUT LIST(SLOPES);
```

prints out all five elements of the array SLOPES.

An array may have any PL/I data type. For example, consider this DECLARE statement,

```
DECLARE NAMELIST(50) CHARACTER(10);
```

which specifies that NAMELIST is to be an array of 50 elements, each of which contains 10 characters.

See Chapter 5 for a more detailed discussion of arrays. For methods of manipulating entire arrays of numbers, see the sections on aggregate expressions in Chapter 6.

Structures

The second form of aggregate is called a structure. The structure is similar to the file definition in the COBOL programming language and to the RECORD data type in Pascal, but a number of important differences exist.

When you use a DECLARE statement to declare an array, all elements of the array must have the same data type. However, the individual data elements of a structure may have different data types. For example, consider the following declaration:

```
DECLARE 1 STUDENT,  
        2 NAME CHARACTER(20),  
        2 QUIZ_AVG FIXED DECIMAL(4, 1),  
        2 EXAM_GRADE FIXED DECIMAL(3),  
        2 LETTER_GRADE CHARACTER(1);
```

STUDENT is a structure aggregate containing four individual data elements, each of which has an additional name as well as a data type. Note that elements of the structure are separated by commas.

You may refer to the individual elements of the structure STUDENT by means of the following names:

```
STUDENT.NAME
STUDENT.QUIZ_AVG
STUDENT.EXAM_GRADE
STUDENT.LETTER_GRADE
```

Therefore, for example, you may set or fetch any of the four individual data elements by using the names as illustrated by the following statements:

```
STUDENT.EXAM_GRADE = 83;
STUDENT.LETTER_GRADE = 'B';
```

These two statements set two of the individual data elements in the structure STUDENT.

If you use the name STUDENT by itself, you are referencing the entire structure aggregate collectively. For example, the statement

```
PUT LIST(STUDENT);
```

prints out all four elements of the structure.

Other Aggregate Options

PL/I supports two aggregate types, arrays and structures. However, you may define aggregates of unlimited complexity by combining arrays and structures in various ways. For example, you may define an array of structures, which is an array each of whose elements is a structure; or you may define a structure whose elements are arrays or other structures. (See Chapter 5.)

PL/I allows you to manipulate aggregates just as you can manipulate ordinary scalar variables. For example, under certain circumstances, you can add an array to a structure and get an array of structures. (See Chapter 6.)

INPUT/OUTPUT

You have already seen several examples of the GET and PUT statements, which perform input from and output to your terminal. You may use different forms of these same statements to perform input/output to files and devices. For example, the arbitrary statement

```
PUT LIST(X + Y);
```

performs output to your terminal, while you might use a statement like

```
PUT FILE(OUTFILE) LIST(X + Y);
```

to perform output to another device or file. For a full description of output to files, see Chapter 11, STREAM INPUT/OUTPUT, and Chapter 12, RECORD INPUT/OUTPUT.

This section is restricted to simple forms of the PUT and GET statement to do output only on your terminal.

The PUT LIST Statement

You may use the form

```
PUT LIST(expression);
```

or

```
PUT LIST(expression, expression, ...);
```

to print the values of one or more expressions on your terminal. Each expression may be any expression, including one of the following:

- A number or a variable. For example, the statement

```
PUT LIST(25, X);
```

prints the value 25, and then the value of X.

- A CHARACTER string constant. For example, the statement

```
PUT LIST('END OF PROGRAM');
```

displays the following output:

```
END OF PROGRAM
```

- An arbitrary PL/I expression. For example, the statement

```
PUT LIST(A * B + C, Q + SQRT(X));
```

causes PL/I to evaluate each of the expressions and print their values.

Before printing the value of each item, PL/I moves to the next of a collection of predetermined tab stop positions on the output line. In Rev. 19, these are every seventh column, starting with column 4.

The SKIP and PAGE Options

PL/I does not automatically move to the next line of output until the current line is filled. Therefore, for example, the two statements

```
PUT LIST(A);
PUT LIST(B);
```

have exactly the same effect as the following single statement:

```
PUT LIST(A, B);
```

In both cases, PL/I prints the values of A and B on the same line.

If you wish to print A and B on separate lines, use the following statements:

```
PUT LIST(A);
PUT SKIP LIST(B);
```

The SKIP option causes PL/I to skip to a new line before printing the value of B.

You may use any one of the following options in the PUT statement:

- SKIP: Skip to a new line before printing.
- SKIP(n): Skip n lines before printing.
- PAGE: Skip to a new page before printing.

The GET LIST Statement

You may use either of the forms

```
GET LIST(variable);
```

or

```
GET LIST(variable, variable, ...);
```

to input the values of various variables from your terminal. When PL/I reaches one of these GET statements, execution of your program stops until you type values for the variables specified.

When you type input to the GET LIST statement, type each input value as a constant. For example, when your program executes a statement like

```
GET LIST(A, B, C);
```

type something like the following as input:

```
25, 67, 43,
```

Other PUT and GET Statement Options

In addition to PUT LIST and GET LIST, PL/I provides the following:

- PUT EDIT, which allows formatted output. This is similar to the formatted output capability provided by the FORTRAN language PRINT statement and the BASIC language PRINT USING statement.
- GET EDIT, which provides formatted input.

- PUT DATA, used mostly for debugging. PL/I prints out each variable name with its value.
- GET DATA, which allows you to specify at execution time which variables you wish to set on input.

See Chapter 11, STREAM INPUT/OUTPUT, for information on the use of these options.

RECORD Input/Output

The PUT and GET statements discussed above are part of the STREAM input/output capability of the PL/I language. When you use these statements, PL/I treats the external file or device as a stream of characters and generally ignores boundaries between records. For a description of the features of STREAM input/output, see Chapter 11.

PL/I provides an additional set of input/output statements, called RECORD input/output. An example of a RECORD input/output statement is

```
READ FILE(TAPE) INTO(S);
```

which could be used to read a tape record into a structure S. For a description of RECORD input/output, see Chapter 12.

OTHER FEATURES OF THE PL/I LANGUAGE

This section summarizes other features of PL/I. For full details, see the referenced sections of this manual.

Subroutines and User-defined Functions

In PL/I terminology, a PL/I program is called a procedure. Remember that your main program begins with a PROCEDURE statement and ends with an END statement.

A user-defined function or subroutine is also called a procedure. Such a function or subroutine also begins with a PROCEDURE statement and ends with an END statement.

A subroutine or function procedure is classified as either internal or external. An external procedure is compiled separately from the main program, and then linked to the main program when the entire program is loaded. (Your main program is also an external procedure.) An internal procedure is nested in your main procedure (or any external procedure), and is compiled at the same time.

For more information on subroutine and function procedures, see Chapter 8.

Error and Condition Handling

A condition is any event that alters the normal execution of your program. Examples of conditions are end of file and program errors.

Using the ON statement, you may specify what action your program may take. For example, you may specify that when end of file occurs, your program should transfer to another section of code, but that on a floating-point arithmetic overflow error your program should print an error message and stop.

By means of condition prefixes, you may control whether PL/I even monitors certain errors. For example, PL/I does not normally check your array subscript values to see if they are in range. But by means of condition prefixes, you may specify that the PL/I compiler is to generate additional code to check for subscript errors.

Chapter 13 contains a general discussion of condition handling. Chapter 12 discusses how you specify action to be taken for input/output conditions, such as end of file.

Block Structuring and Storage Allocation

PL/I is a block-structured language, meaning that each program consists of blocks of code, and that declarations are local to one or more of these blocks of code. An example of a program block is a procedure, mentioned above. For a discussion of the rules, see Chapter 9.

Related to the declaration of variables is the question of when storage for variables is allocated. PL/I gives you a great deal of control over management of your storage allocation. (See Chapter 7.)

5

Data Types and Data Attributes

DATA TYPES: INTRODUCTION

In PL/I every constant and variable has both a value and a data type. This chapter describes the major kinds of PL/I data types -- arithmetic, string, and pictured — and explains how to specify constants and variables for each data type. The chapter also describes aggregates. An aggregate may be either an array of values with the same data type, or a structure of values with possibly different data types. Finally, the chapter introduces the attributes of data types: the ways in which you may specify how a constant or variable is stored in memory.

An analysis of the following assignment statement demonstrates the relationship between the value and the data type of variables and constants:

```
FAHRENHEIT = 1.8 * CELSIUS + 32;
```

This statement uses two variables, FAHRENHEIT and CELSIUS, and two constants, 1.8 and 32, to specify the familiar formula for converting a temperature in degrees Celsius to degrees Fahrenheit.

Values of Variables and Constants: The value of a constant is determined by the constant itself; for example, the constant 1.8 has the value 1.8. The value of a constant remains the same throughout execution of the PL/I program. On the other hand, the value of a variable changes during execution of the program. For example, after

the above statement has been executed, the value of the variable FAHRENHEIT would be changed to the value of the expression on the right-hand side of the assignment statement.

Data Types of Constants: The constant 32 has the data type FIXED DECIMAL(2). This means that 32 contains two decimal digits. On the other hand, the constant 1.8 has the data type FIXED DECIMAL(2,1), indicating that the constant contains two decimal digits, one of which follows the decimal point.

Data Types of Variables: You may specify the data type of a variable by means of a DECLARE statement. For example, suppose that the program containing the assignment statement shown above also contains the following DECLARE statement:

```
DECLARE FAHRENHEIT FIXED DECIMAL(6,2);
```

This DECLARE statement specifies that the variable FAHRENHEIT occupies a storage area large enough to accommodate six decimal digits, and that two of these digits follow the decimal point. This means that the variable FAHRENHEIT can be assigned any value from -9999.99 to 9999.99, inclusive.

The keyword FIXED in the data type of FAHRENHEIT indicates that the position of the decimal point in the digits representing FAHRENHEIT is fixed; that is, FAHRENHEIT has a value of six digits, and two of these digits always follow the decimal point. If you DECLARE a variable to be FLOAT, rather than FIXED, you are specifying that the decimal point may be in any position with respect to the digits in the value. For example, suppose your program contained the following declaration:

```
DECLARE CELSIUS FLOAT DECIMAL(4);
```

This statement specifies that the variable CELSIUS occupies a storage area large enough to hold four decimal digits, and that the decimal point can appear anywhere with respect to those digits. For example, CELSIUS could have the value 8.264×10^{12} .

Data Types: Classification

Below is a summary of the two major classes of PL/I data types: computational data types and noncomputational data types. Computational data types specify values that can be used in computations, such as addition or multiplication. Noncomputational data types specify values on which no such computations can be made.

Computational Data Types

There are three groups of computational data types in PL/I, all of which are described in detail in this chapter:

- Values of the arithmetic data type are ordinary numbers that can be combined by the usual arithmetic operations, such as addition, subtraction, multiplication, and division.
- Values of the string data type are sequences of characters or bits. PL/I provides operators and functions that permit you to pull strings apart and put them together, as well as to perform various string searches and translations.
- Values of the pictured data type are also sequences of characters, but such sequences are constrained in various ways. For example, you may declare a variable to be a pictured value whose string of characters can contain only digits and a decimal point.

Noncomputational Data Types

Noncomputational data types are used for program control, input/output control, and storage control. The noncomputational data types and the chapters where they are discussed are as follows:

- A value of the LABEL data type is the label of a PL/I statement (4, 7, 10).
- A value of the FORMAT data type is the label of a FORMAT statement (11).
- A value of the ENTRY data type is the entry point to a procedure (8, 10).
- A value of the FILE data type is an identifier associated with a file (12).
- A value of the POINTER data type is the storage address of data (7).
- A value of the AREA data type is a block of storage that you can sub-allocate to store other variables (7).
- A value of the OFFSET data type is the displacement of a storage block within a given AREA (7).

ARITHMETIC DATA TYPES: INTRODUCTION

PL/I has many arithmetic data types. Each of these data types has a base, scale, mode, and precision, and in the case of FIXED arithmetic data types, a scale factor. These terms have the following meanings:

Base: The base of an arithmetic data type is either DECIMAL or BINARY. Your choice of base indicates whether you want PL/I to manipulate the data internally as a binary number or a decimal number. Most of the time it does not make any difference whether you use DECIMAL or BINARY, since you usually get the same answers either way. In fact, for FLOAT data there is no difference whatsoever in the internal representation. However, for FIXED, there are some differences: FIXED BINARY is usually more efficient than FIXED DECIMAL when the values being represented are integers (that is, the scale factor is 0). On the other hand, FIXED DECIMAL is usually more accurate than FIXED BINARY if the values are not integers, such as in dollars and cents computations.

Note

BINARY is used only with arithmetic data types, and is different from the BIT data type, which is specified for certain types of strings.

Scale: The scale of arithmetic data is either FIXED or FLOAT. Data having the FIXED attribute has its decimal point (or binary point) at a fixed position with respect to the digits in the value. Data with the FLOAT attribute can have its decimal (or binary) point in any position with respect to the digits in the value. Ordinarily, use FIXED when your data is going to be integers, or in commercial applications when you are dealing with dollars and cents values that must be accurate to the penny and that are within well-defined ranges. Ordinarily, use the FLOAT data type in scientific applications when you are interested in accuracy to a certain number of significant digits, regardless of where the decimal point is in relation to those digits.

Mode: The mode of arithmetic data is either REAL or COMPLEX. You will nearly always use REAL. Use the COMPLEX data type only in those engineering or mathematical applications where it is necessary to manipulate imaginary or complex numbers.

Precision: The precision of arithmetic data of scale FIXED is the maximum number of digits in the value of the data. If the scale is FLOAT, the precision is the number of significant digits in the mantissa. If the base is DECIMAL, the precision is the number of decimal digits; if the base is BINARY, the precision is the number of binary digits or bits. PL/I allocates to the data element a block of storage that is large enough to accommodate the number of digits specified by the precision, as well as a sign.

Scale Factor: The scale factor is applicable only to FIXED data, and it specifies the number of digits following the decimal point or binary point. If the scale factor is 0, the data value is always an integer. If the scale factor is positive, it specifies the number of digits to the right of the decimal point. If the scale factor is negative, it specifies that the rightmost digit in the value is one or more positions to the left of the implied decimal or binary point.

The next few sections cover the arithmetic data types in more detail.

Arithmetic Data Types: FIXED DECIMAL REAL

The simplest kind of FIXED DECIMAL constant consists of one or more decimal digits, possibly with a decimal point, possibly with a sign. The precision of the constant is the number of digits, and the scale factor of the constant is the number of digits following the decimal point. If there is no decimal point, or if no digits follow the decimal point, the scale factor is 0.

For example, the constant 23 has the attribute FIXED DECIMAL(2,0); that is, 23 is FIXED DECIMAL with a precision of 2 and a scale factor of 0. The constant 894.7 and the constant -482.3 each have the attributes FIXED DECIMAL(4,1); that is, a precision of 4 and a scale factor of 1. The constant .897 has a precision of 3 and a scale factor of 3, since all digits follow the decimal point.

It is possible to have two FIXED DECIMAL constants with the same value but with different precisions or scale factors. For example, the constants 23, 023, and 23.0 all have the same value (23), but they have precisions of 2, 3, and 3, respectively, and scale factors of 0, 0, and 1, respectively.

PL/I permits you to form a more complex type of FIXED DECIMAL constant by appending to it the letter F followed by a decimal number (optionally signed) that specifies a power of ten by which the value of the decimal number is to be multiplied. For example, the constant 23.892F1 has the value 238.92 (since F1 specifies that the value is to be multiplied by ten), has a precision of 5 (since there are five digits in the value portion of the constant, and since the digit 1 following the F does not contribute to the precision), and has a scale factor of 2 since the last two digits of the value follow the decimal point.

A more complicated example is the constant 23F4. This constant has value 230000, but has the attribute FIXED DECIMAL(2,-4), for the following reasons: the precision is 2 because there are two digits in the value portion of the constant 23F4. The scale factor is -4, since the value digits in the constant (23) are four positions to the left of the decimal point. Similarly, the constant 46.2F20 has the attributes FIXED DECIMAL(3,-19), and has the value 46.2×10^{20} .

The number following the F may be negative, in which case it means that the decimal point is to be moved to the left. For example, the constant 23.892F-1 has the value 2.3892, but has the attributes FIXED DECIMAL(5,4), since four decimal digits in the value follow the decimal point. The constant 23F-4 has the value .0023 and the attributes FIXED DECIMAL(2,4); the precision is 2 because there are two digits (23) in the value, and the scale factor is 4 because there are four digits following the decimal point. Similarly, the constant 23.8F-20 has the attributes FIXED DECIMAL(3,19), and the value 23.8×10^{-20} .

You may use the DECLARE statement to specify that a variable is to be FIXED DECIMAL and, at the same time, you may specify the precision and scale factor of the variable. The simplest form of such a declaration is

```
DECLARE variable FIXED DECIMAL(p,q);
```

which indicates that the specified variable is FIXED DECIMAL, with a precision of p and a scale factor of q. The precision of p indicates that the variable occupies a storage block large enough to accommodate p decimal digits and a sign. The scale factor of q indicates that q of these digits follow the implied decimal point.

For example, if your program contains the statement

```
DECLARE SALARY FIXED DECIMAL(7,2);
```

then you are telling PL/I that the variable SALARY is to have seven decimal digits, two of which follow the decimal point. This means that SALARY can have a value as large as 99999.99 or a negative value as small as -99999.99.

If the scale factor is 0, you need not specify the scale factor in the declaration. For example, the statement

```
DECLARE INDEX FIXED DECIMAL(3);
```

says that INDEX can have a value between -999 and +999, with no digits following the decimal point (the scale factor is 0).

It is possible for a scale factor to be negative. In this case, the decimal point is to the right of the rightmost digit in the stored value of the variable. For example, if a variable has the attributes FIXED DECIMAL(2,-3), the variable may be assigned any value between -99000 and +99000, as long as the value is a multiple of 1000. For example, you may assign the value 86000 to such a variable, in which case PL/I stores the digits 86 and remembers that the decimal point is three positions to the right of the 6.

It is also possible for the scale factor to be positive and larger than the precision. In this case, the decimal point is one or more positions to the left of the leftmost position in the stored value of the variable. For example, if a variable has the attributes FIXED DECIMAL(2,5), it can take on any value between -.00099 and +.00099. If you assigned the value .00086 to such a variable, PL/I would store the two digits 86, and would remember that the decimal point was five positions to the left of the 6.

Your program may, of course, assign any numeric value to any FIXED DECIMAL variable. If the value being assigned has the same precision and scale factor as the variable to which it is being assigned, PL/I can perform the assignment directly. However, it is possible that the value being assigned has a different precision or scale factor. In this case, PL/I must modify the value being assigned before the operation can be completed. Usually, this modification is precisely what you would expect.

Consider, for example, the variable SALARY declared above as FIXED DECIMAL(7,2). If a program contains the assignment statement

```
SALARY = 12345.67;
```

then PL/I can make the assignment directly, since the value being assigned is also FIXED DECIMAL(7,2). On the other hand, if the program contains the statement

```
SALARY = 23;
```

then PL/I actually assigns the value 00023.00 to SALARY. If the value being assigned has extra digits to the right of the decimal point, PL/I throws these additional digits away, or truncates the value.

For example, if your program contains the statement

```
SALARY = 482.937;
```

then PL/I assigns the value 00482.93 to SALARY. Notice that PL/I does not round to 482.94; the value being assigned is truncated, meaning that extra digits on the right are thrown away. Finally, if the value being assigned has too many digits to the left of the decimal point, the assignment produces a SIZE error. The assignment statement

```
SALARY = 1234567;
```

is an example of a SIZE error in the assignment.

PL/I Reference Guide

As another example, suppose that the variable PART has the attributes FIXED DECIMAL(2,-3). Then PART can only have values that are multiples of 1000, and these values must lie between -99000 and +99000. If your program attempts to assign a value greater than 99000 or smaller than -99000 to the variable PART, the assignment produces a SIZE error. On the other hand, if your program contains the assignment

```
PART = 43827;
```

then PL/I truncates the last three digits, with the result that PART is assigned the value 43000.

In the preceding paragraphs, only assignment of constants to FIXED DECIMAL variables has been discussed. The same rules apply, however, whenever a variable or expression is assigned to a FIXED DECIMAL variable.

For example, suppose that the variable A is FIXED DECIMAL(7,2), and the variable B is FIXED DECIMAL(4,1). Suppose that your program contains the assignment statement

```
A = B;
```

Since A has five digits to the left of the decimal point, and B has three digits to the left of the decimal point, this assignment cannot produce a SIZE error. Furthermore, since A has two digits to the right of the decimal and B has one digit to the right of the decimal point no truncation takes place. On the other hand, suppose that your program contains the assignment statement

```
B = A;
```

How PL/I handles this depends upon the value of A. As long as A is less than 999 and greater than -999, this assignment is legal; but if A is outside of that range, the assignment causes a SIZE error. If the assignment is legal, B is assigned the value of A truncated after the first digit to the right of the decimal point.

For maximum precision, default precision, and the maximum and minimum allowable scale factor of FIXED DECIMAL REAL numbers, see Appendix C.

Arithmetic Data Types: FLOAT DECIMAL REAL

Data with the FIXED attribute have their decimal point in a fixed position relative to the digits of the value; the scale factor is used to specify that position. With FLOAT data, there is no scale factor, since the decimal point may be in any position with respect to the digits of the value.

A FLOAT DECIMAL constant contains one or more decimal digits (optionally with a decimal point, and optionally preceded by a minus sign) followed by the letter E followed by an (optionally signed) decimal number. The value to the left of the letter E is called the mantissa of the FLOAT constant, and the number to the right is called the exponent or characteristic. The precision of the FLOAT constant is the number of digits in the mantissa.

For example, the constant 2.34E+05 is a constant with the attributes FLOAT DECIMAL(3). The mantissa is 2.34, and the exponent is 5. The precision is 3, since there are three digits in the mantissa. The value of the constant is 2.34×10^5 . The constant 2.34E+05 has exactly the same value as 2.34F+05, but the latter has the attributes FIXED DECIMAL(3,-4). This means that, although the two constants have the same value, PL/I uses different methods to represent the constants internally in your PL/I program. Use of the different constants may produce different results.

As another example, the constant 894.267E-4 has the attributes FLOAT DECIMAL(6), and has the same value as the constant .0894267. However, the latter constant has the attributes FIXED DECIMAL(7,7) and so has a different internal representation. In fact, any FIXED DECIMAL constant can be rewritten as a FLOAT DECIMAL constant with the same value. For example, the constant 23, which has the attributes FIXED DECIMAL(2), can be rewritten as 23E+00 with the attributes FLOAT DECIMAL(2), or as 023, or as 023E+00 with the attributes FLOAT DECIMAL(3). Or, the constant 8.94E+00, with the attributes FIXED DECIMAL(3,2), can be rewritten as .894E1, with the attributes FLOAT DECIMAL(3).

You may use the DECLARE statement to specify that a variable is FLOAT DECIMAL. For example, the statement

```
DECLARE SCOPE FLOAT DECIMAL(8);
```

specifies that the variable SCOPE is a FLOAT variable occupying enough storage to accommodate eight decimal digits, or significant digits. The decimal point may appear in any position with respect to these digits.

For maximum exponent range and maximum precision of FLOAT DECIMAL REAL numbers, see Appendix C.

Arithmetic Data Types: FIXED BINARY REAL

PL/I gives the programmer the option of storing constants and variables using the binary, rather than the decimal, number base. Internally, this means that PL/I stores your FIXED data as two's-complement binary numbers, rather than the binary-coded decimal format used with the DECIMAL attribute.

The simplest form of FIXED BINARY constant consists of a string of binary digits (0's and 1's), with an optional sign and an optional binary point. (The term binary point is used in the binary number system in the same way that the decimal point is used in the decimal number system; in both cases, the function is to separate the integer digits from the fractional digits.) The constant must end with the letter B to indicate that it is a BINARY constant. For example, the constant 10110B is a FIXED BINARY constant that has the same value as the FIXED DECIMAL constant 22.

The terms precision and scale factor are used for FIXED BINARY constants and variables in the same way as for all FIXED DECIMAL constants and variables, except that now the phrase "number of digits" refers to binary digits, rather than decimal digits. For example, the constant 1001.010B is a constant with the attributes FIXED BINARY(7,3); the precision is 7 because there are seven binary digits, and the scale factor is 3 since three of the digits follow the binary point. This constant has the same value as the constant 9.25, which has the attributes FIXED DECIMAL(3,2).

As in the case of FIXED DECIMAL constants, a FIXED BINARY constant may contain the letter F, followed by a decimal constant, to specify the number in decimal that a binary point should be moved to the right or left. For example, the constant 11.01101F3B has the data type FIXED BINARY(7,2), and has the same value as 11011.01B. (In decimal, this value could be written as 27.25.) Similarly, the constant 11011F5B has the data type FIXED BINARY(5,-5), and has the value 1101100000B (equal to decimal 864), which has the attributes FIXED BINARY(10,0).

Notice, in particular, that the value following the letter F is a decimal constant, not a binary constant. As another example, the constant 1101F-25B has the attributes FIXED BINARY(4,25), and has a value equal to $13 \times 2^{(-25)}$.

Further examples of FIXED BINARY constants are shown in Table 5-1. This table illustrates various FIXED BINARY constants and indicates their precisions and scale factors, as well as the values of the constants in decimal.

Table 5-1
Examples of FIXED BINARY Constants

Constant	Precision	Scale Factor	Value
101B	3	0	5
00101B	5	0	5
101.1B	4	1	5.5
101.100B	6	3	5.5
000101.100B	9	3	5.5
10100B	5	0	20
101F2B	3	-2	20
.101F5B	3	-2	20
.10100F5B	5	0	20
100.0F-3B	4	4	.5
10F-3B	2	3	.25
1101F-20B	4	20	13×2^{-20}

You may use the DECLARE statement to specify that a variable is to have FIXED BINARY attributes. For example, if the statement

```
DECLARE DATUM FIXED BINARY(3,0);
```

appears in your program, it specifies that DATUM is a variable for which the compiler should allocate enough storage to accommodate three binary digits and a sign. This means that DATUM can have values between -111B, or -7, and 111B, or 7. Since the scale factor is 0, DATUM can have any integer value between those two values.

In the declaration of a FIXED BINARY variable, the precision specifies the number of binary digits in the value of the variable, and the scale factor specifies the number of those digits that lie to the right of the binary point. For example, if the statement

```
DECLARE LINK FIXED BINARY(5,2);
```

appears in your program, it specifies that LINK is a variable that can hold five binary digits, and that two of these lie to the right of the binary point. This means that LINK can have values as large as 111.11B (or 7.75), and as small as -111.11B (or -8.00). It can take on any value between these two extremes, only in increments of .01B (or .25).

This means that the variable can take on such values as 3.25, 1.75, and -6.5.

If the scale factor is negative, it specifies the number of implied zeros that lie to the right of the rightmost digit in the value of the variable. For example, if your program contains the DECLARE statement

```
DECLARE LONGITUDE FIXED BINARY(3,-4);
```

then LONGITUDE has enough storage allocated to hold three binary digits and a sign, with four implied zeros following the rightmost of these digits. This means that LONGITUDE can have a value as large as 1110000B, or as small as -1110000B, in increments of 10000B. (Writing these figures in decimal, the variable can have values as large as 112 or as small as -128, in increments of 16. This means that LONGITUDE can take on any of the following values, and only the following values: 112, 96, 80, 64, 48, 32, 16, 0, -16, -32, -48, -64, -80, -96, -112, and -128).

Examples of FIXED BINARY variable data types are shown in Table 5-2.

Table 5-2
Examples of FIXED BINARY Variable Data Types

Data Type of Variable	Range of Values:		In Increments of
	From	To	
FIXED BIN (3,0)	-1000B (=-8)	111B (=7)	1
FIXED BIN (7,0)	-100000001B (=-128)	1111111B (=127)	1
FIXED BIN (5,2)	-1000.00B (=-8)	111.11B (=7.75)	.01B (=.25)
FIXED BIN (3,4)	-.0111B	+.0111B	.0001B
FIXED BIN (3,-4)	-1110000B (=-112)	+1110000B (=-112)	10000B (=16)

As in the case of assignments to FIXED DECIMAL variables, if the value assigned to a FIXED BINARY variable does not match the data type of the target variable, PL/I changes the value being assigned to match the target variable. If the value being assigned has too many digits to the right of the binary point, the extra digits are truncated. If there are too many digits to the left of the binary point, a valid assignment cannot be made, and a SIZE error occurs. Table 5-3 illustrates such assignments.

Table 5-3
Examples of FIXED BINARY Assignments

Data Type of Target Variable	Value of Expression Being Assigned	Value Actually Assigned as a Result
FIXED BIN (3,0)	4	4
FIXED BIN (3,0)	4.9	4 (truncation)
FIXED BIN (3,0)	-4.7	-4 (truncation)
FIXED BIN (3,0)	12	SIZE error
FIXED BIN (3,0)	5	5 (=101.00B)
FIXED BIN (15,2)	5.43	5.25 (=101.01B) truncation
FIXED BIN (15,2)	1101.1011B	1101.10B (truncation)
FIXED BIN (15,-2)	1110B	1100B (truncation)
FIXED BIN (15,-2)	87	84 (truncation)

Maximum precision and maximum scale factor for FIXED BINARY REAL numbers are given in Appendix C.

Arithmetic Data Types: FLOAT BINARY REAL

FLOAT BINARY constants are written as a string of binary digits (zeros and ones), optionally with a sign and a binary point, followed by the letter E and then by a decimal number indicating the number of positions that the binary point is to be shifted relative to the binary digits in the value. The entire constant is ended with the letter B. The terminology used with FLOAT BINARY values is similar to that used with FLOAT DECIMAL values. The value to the left of the letter E is called the mantissa of the FLOAT value, and the decimal constant to the right of the letter E is called the exponent or characteristic of the value. The precision of the FLOAT BINARY constant is the number of digits in the mantissa. There is no scale factor in FLOAT values, since the position of the binary point, as specified by the characteristic, may be in any position relative to the digits in the mantissa.

For example, the constant 101E3B is a FLOAT BINARY(3) constant. The precision is 3 since there are three digits in the mantissa. This constant has the same value as the constant 101000B, but the latter has the attributes FIXED BINARY(6,0). Each of these two BINARY constants has the same value as the DECIMAL constant 40, the latter having the attributes FIXED DECIMAL(2,0).

A negative exponent indicates the number of positions that the binary point is to be moved to the left with respect to the value in the mantissa. For example, the constant 1011E-2B has the attributes FLOAT BINARY(4). It has the same value as the constant 10.11B, which has the attributes FIXED BINARY(4,-2). It also has the same value as the constant 2.75, which has the attributes FIXED DECIMAL (3,2).

Table 5-4 illustrates several FLOAT BINARY constants, giving their precisions and values.

Table 5-4
Examples of FLOAT BINARY Constants

Constant	Precision	Value
101E0B	3	5
101E3B	3	40
000101E3B	6	40
101.000E3B	6	40
101E45B	3	5×2^{45}
1011E-2B	4	2.75
11011E-23B	5	27×2^{-23}

You may use the DECLARE statement to specify that a variable is to be FLOAT BINARY. For example, if the statement

```
DECLARE RANGE FLOAT BINARY(47);
```

appears in your program, you are specifying that the variable RANGE occupies enough storage to accommodate 47 significant binary digits (bits), with the binary point in any position with respect to those digits. Maximum precision of FLOAT BINARY REAL numbers is given in Appendix C.

Arithmetic Data Types: COMPLEX

Up until now, all the arithmetic data types that have been considered have been REAL, meaning real in the mathematical sense of not using complex or imaginary numbers (numbers that are expressed using the square root of -1). If you are writing a mathematical or engineering application requiring the use of imaginary or complex numbers, use the COMPLEX data type supplied by the PL/I language.

In preceding sections, several different kinds of arithmetic constants are described. All of these constants have REAL attributes, since REAL is the default mode for arithmetic constants. If you take any of these constants and append the letter I to the end, the result is a constant that PL/I recognizes as representing an imaginary value.

Table 5-5 illustrates a number of COMPLEX constants. Each of these constants is identical to one in a preceding section, except that the letter I has been added to the end. The data type of the constant is derived as in preceding sections with the precision represented by the number of digits, and, for FIXED constants, the scale factor representing the number of digits following the decimal point or binary point. The only difference in the data type is in the attribute COMPLEX, where, in preceding sections, the constant has the implied attribute REAL. The values of these constants are imaginary in the mathematical sense, with each constant represented as a multiple of i, the square root of -1.

Table 5-5
Table of COMPLEX Constants

Line #	Constant	Data Type	Value
1	23I	FIXED DECIMAL (2,0) COMPLEX	23i
2	86.45F3I	FIXED DECIMAL (4,-1) COMPLEX	86450i
3	45E0I	FLOAT DECIMAL (2) COMPLEX	45i
4	1101BI	FIXED BINARY (4,0) COMPLEX	13i
5	10F-4BI	FIXED BINARY (2,4) COMPLEX	.125i
6	10E-4BI	FLOAT BINARY (2) COMPLEX	.125i

If you declare a variable to be COMPLEX, you are specifying that the storage allocated for the variable is large enough to hold two numbers, one for the real part of the value of the variable, and one for the imaginary part. For example, if the statement

```
DECLARE STRENGTH FIXED DECIMAL(10,2) COMPLEX;
```

appears in your program, STRENGTH is a variable whose storage area can accommodate two numbers, one of which is the real part of the value of STRENGTH, and one of which is the imaginary part; each of these numbers can hold ten decimal digits, two of which follow the decimal point.

Your program may contain expressions involving COMPLEX constants and variables. PL/I can evaluate these expressions according to the mathematical rules for complex numbers to achieve COMPLEX results. If

a COMPLEX expression is assigned to a REAL variable, the real part of the expression is assigned to the target variable, and the imaginary part of the value of the expression is discarded. If a COMPLEX expression is assigned to a COMPLEX variable, the real part of the value of the expression is assigned to the real part of the target variable, and the imaginary part of the expression is assigned to the imaginary part of the variable.

Finally, if a REAL expression is assigned to a COMPLEX variable, the value of the expression is assigned to the real part of the variable, and the value 0 is assigned to the imaginary part.

For assigning the real part or the imaginary part of a value, the rules for truncation and SIZE errors are the same as those given above in the sections on real constants. For example, if the program with the declaration of STRENGTH contains the following assignment statement

```
STRENGTH = 23 + 45.789I;
```

then the last digit of the imaginary part of the value is truncated, with the result that STRENGTH is assigned the value 23 + 45.78I.

Declarations of Arithmetic Variables

Use the DECLARE statement to specify the attributes of an arithmetic variable. Every arithmetic variable has a base, scale, mode, and precision. Furthermore, if the variable is FIXED, a scale factor is specified as part of the precision. If you omit one of these from the list of attributes in the DECLARE statement, PL/I supplies a default. The rules for base, scale, and mode are as follows:

- The base is either BINARY or DECIMAL. If you prefer, you may use the abbreviations BIN or DEC, respectively. If your DECLARE statement specifies no base, the default of BINARY is supplied.
- The scale is either FIXED or FLOAT. If the DECLARE statement does not specify a scale, PL/I supplies a default of FIXED.
- The mode is either REAL or COMPLEX. If you prefer, you may use the abbreviation CPLX for COMPLEX. If you do not specify a mode, PL/I supplies a default of REAL.

In your DECLARE statement, you may specify the base, scale, and mode in any order. Specify the precision, and for FIXED variables the scale factor, in parentheses following any one of the base, scale, or mode attributes. If you do not specify a scale factor for a FIXED variable, PL/I supplies a default of 0. You may, if you prefer, specify the precision with a separate keyword PRECISION (abbreviation PREC). For example, all of the following five declarations are equivalent:

```
DECLARE VALUE FIXED DECIMAL(5) REAL;
DECLARE VALUE FIXED(5) DEC REAL;
DECLARE VALUE DEC(5,0);
DECLARE VALUE FIXED DECIMAL REAL PREC(5,0);
DECLARE VALUE DEC PRECISION(5);
```

These five DECLARE statements all give the same attributes of VALUE. The attributes FIXED and REAL need not be specified, since they are the defaults. The precision can be specified along with any of the keywords, or with a separate PRECISION keyword. The scale factor of 0 need not be specified, since 0 is the default.

Summary of Arithmetic Constants

Below is a brief summary of the rules for formation of arithmetic constants, and for deriving their data types.

An arithmetic constant consists of a number of characters, one immediately following the other, with no blanks. These characters are as follows:

1. An optional plus sign or minus sign.
2. A string of digits, optionally containing a radix point. If the base of the constant is BINARY, the digits may be only zeros and ones.
3. An optional F or E, followed by a string of decimal digits. The decimal digits may be preceded optionally by a plus sign or minus sign.
4. An optional B.
5. An optional I.

All of the elements shown above are optional, except those listed in Rule 2. That is, a constant must contain at least one digit.

The data type of the constant is derived according to the following rules:

1. If the constant contains the letter B, the base is BINARY; otherwise the base is DECIMAL.
2. If the constant contains the letter E, the scale is FLOAT; otherwise, the scale is FIXED.
3. If the constant contains the letter I, the mode is COMPLEX; otherwise, the mode is REAL.
4. The number of digits in the precision is the total number of digits appearing in that section of the constant specified by Rule 2 for number of characters above.
5. If the constant is FIXED, the scale is determined as follows:
 - o If the constant contains a radix point, let x equal the number of digits to the right of the radix point; otherwise, let $x = 0$.
 - o If the constant contains the letter F, let y equal the value of the decimal constant following the letter F; otherwise, let $y = 0$. (It is possible for y to be negative.)
 - o The scale factor of the constant is $(x - y)$.

STRING DATA TYPES: INTRODUCTION

String values are sequences of characters. PL/I contains two string data types, CHARACTER strings and BIT strings. CHARACTER strings are useful for representing text; BIT strings are used to represent Boolean or logical values and to manipulate the internal representation of other data types.

In many ways, BIT strings work the same way in PL/I as CHARACTER strings; in fact, the PL/I language was designed to make these two types of strings as similar as possible. The properties shared by CHARACTER and BIT strings include the following: NONVARYING strings of constant length, VARYING strings of variable length, and concatenation and various built-in functions to put strings together and pull them apart.

The two major differences between BIT and CHARACTER strings are as follows:

- The individual elements of a CHARACTER string can be any of the 256 characters supported by PRIME hardware. On the other hand, the individual elements of a BIT string may have only two possible values: a 0-bit or a 1-bit.

- In the internal representation of CHARACTER strings, each character in the string occupies one byte of memory. Each individual element of a BIT string occupies only one bit of memory, making it possible to store eight bit values in a single memory byte.

String Data Types: CHARACTER NONVARYING

The simplest form of CHARACTER constant consists of an apostrophe followed by a string of characters and another apostrophe. The characters between the apostrophes make up the value of the constant. For example, 'ABCDE' is an example of a CHARACTER constant containing the five characters ABCDE.

Use the CHARACTER attribute in a DECLARE statement to specify that the value of the variable is to be a string of characters, rather than a number. Follow the keyword CHARACTER with a parenthesized number to indicate the precise number of characters in the variable. For example, the statement

```
DECLARE NAME CHARACTER(8);
```

specifies that the variable NAME can have, as its value, a string of precisely eight characters. This means that you may assign a string of eight characters to the variable NAME. For example, the assignment statement

```
NAME = 'JOHNSTON';
```

means that the variable NAME has as its value the eight characters in JOHNSTON.

The value of the variable NAME must contain precisely eight characters, no more and no less. If you assign to NAME a character string with a different number of characters, PL/I must modify the value before assigning it to NAME.

If the character string assigned to NAME contains less than eight characters, PL/I pads the value by adding blank characters to it in assigning the value to NAME. For example, if your program executes the assignment statement

```
NAME = 'JONES';
```

then PL/I pads the value being assigned with three blanks so that, after the assignment, the variable NAME has the value 'JONESbbb', where

the symbol b indicates a blank, a nonprinting character. As another example, if your program executes the assignment statement

```
NAME = 'KU';
```

then PL/I assigns the value 'KUbubbbb' to NAME.

If your program assigns to NAME a value containing more than eight characters, PL/I truncates the value by removing from it all characters following the eighth. For example, if your program executes the statement

```
NAME = 'ALEXANDERSON';
```

then PL/I truncates the value by discarding the last four characters, so that the value assigned to NAME is 'ALEXANDE'.

Since the variable NAME must always have precisely eight characters as its value, it is said that NAME has the CHARACTER NONVARYING attributes. For example, the statement

```
DECLARE ADDRESS CHARACTER(30) NONVARYING;
```

specifies that ADDRESS is a variable containing precisely 30 characters, no more and no less. If you do not specify VARYING or NONVARYING in the DECLARE statement, PL/I automatically provides the default attribute NONVARYING.

Under certain circumstances, you may use an arbitrary expression in parentheses following CHARACTER. For example, the statement

```
DECLARE PRODUCT CHARACTER(X + Y);
```

is legal under certain circumstances. These circumstances are described in the section Variables in Extent and INITIAL Expressions in Chapter 7.

String Data Types: CHARACTER VARYING

The variables described in the preceding section have values that must always contain the same number of characters. It is also possible to declare that a variable can contain a varying number of characters.

For example, if your program contains the statement

```
DECLARE CITY CHARACTER(8) VARYING;
```

then CITY is a variable whose value is a string containing eight or fewer characters. That is, CITY can have as its value a string containing 0, 1, 2, 3, 4, 5, 6, 7, or 8 characters, but no more than eight characters. Thus, if your program executes the statement

```
CITY = 'BOSTON';
```

then the value 'BOSTON' is actually assigned to CITY; PL/I does not pad with blanks in this case. On the other hand, if your program assigns a string longer than eight characters to CITY, truncation still takes place. For example, if your program executes the statement

```
CITY = 'PHILADELPHIA';
```

then the last four characters of the value being assigned are discarded, and the truncated value 'PHILADEL' is assigned to CITY.

String Data Types: CHARACTER Constants

The simplest form of a CHARACTER constant is a string of characters between two apostrophes. But PL/I has several other rules for forming CHARACTER constants.

The discussion in this section refers to Table 5-6. Line number 1 of that table illustrates the simple form of CHARACTER constant that has already been discussed. The constant 'ABCDE' contains the five characters ABCDE.

Table 5-6
CHARACTER Constants

Line #	CHAR Constant	Length (# of Characters)	Characters
1	'ABCDE'	5	ABCDE
2	''	0	(none)
3	'DOESN'T'	7	DOESN'T
4	''''	1	'
5	(6) 'A'	6	AAAAAA
6	(4) 'ABC'	12	ABCABCABCABC

The Null String: A null string is a string that contains no characters at all. A null string constant consists of two apostrophes with no characters between them.

When the null string is assigned to a CHARACTER NONVARYING variable, PL/I pads the null string with blanks, so that the variable receives a value of a string of blanks. If the null string is assigned to a CHARACTER VARYING variable, no padding takes place, and the value of the variable is itself the null string. For example, consider the following statements:

```
DECLARE NAME CHARACTER(8);
DECLARE CITY CHARACTER(8) VARYING;
NAME = '';
CITY = '';
```

After these statements have been executed, the variable NAME has the value 'bbbbbbb'. The variable CITY has the value '', the null string.

Using an Apostrophe in a CHARACTER Constant: Since you may wish to include an apostrophe in a CHARACTER constant, PL/I provides a special rule for doing this, as illustrated in line 3 of the table. To represent a single apostrophe within a CHARACTER constant, write two apostrophes. For example, the two apostrophes between the letters N and T in the constant 'DOESN'T' are recognized by PL/I as representing a single apostrophe, so that the CHARACTER constant really contains only seven characters, DOESN'T.

Line 4 of the table illustrates a very special case of this. The string contains exactly one character, that character being an apostrophe. This CHARACTER constant is written ''''.

Repetition Factors: If your CHARACTER constant is made up of one or more characters repeated over and over, then, for convenience, you can use a repetition factor. The repetition factor is a decimal integer, with no sign, enclosed in parentheses just before the first apostrophe in the constant. For example, (6)'A' is a CHARACTER constant containing six characters, AAAAAA. This constant could also have been written as 'AAAAAA'; PL/I treats these two constants in exactly the same way.

Line number 6 of the table illustrates the case where several characters are being repeated. The constant (4)'ABC' contains twelve characters, ABCABCABCABC. This constant could also have been written 'ABCABCABCABC'.

String Data Types: BIT NONVARYING and VARYING

The CHARACTER data types that have been described in the last few sections are only one form of PL/I's string data types. The second form of string data type is called BIT.

Note

Do not confuse the BIT data type with the BINARY data type. BINARY variables are numeric, and have values that are numbers. BIT variables are string variables, and their values are strings of bits.

The BIT data type is used far less often by programmers than the CHARACTER data type. The two main uses of the BIT data type are as follows:

- Other languages have a so-called logical or Boolean data type, which the programmer uses to manipulate values that are either true or false. In PL/I, the role of the logical data type is played by a string declared as BIT(1). A BIT(1) string contains one individual element, which is either a 0-bit or a 1-bit. The 1-bit represents true, and the 0-bit represents false.
- The PL/I language was designed so that a programmer taking reasonable precautions can write a program in an implementation-independent manner. This means that if the program runs on two different machines, it should give the same answers. However, sometimes a programmer needs to write a machine-dependent program that examines and manipulates the bit formats of other data elements. PL/I allows this with the UNSPEC built-in function and pseudo-variable, which are described in Chapter 14.

PL/I Reference Guide

The simplest form of a BIT constant is an apostrophe ('), followed by a string of zeros and ones, followed by another apostrophe, followed immediately by the letter B. For example, '10110'B is a BIT constant containing five bit values, a 1-bit followed by a 0-bit followed by two 1-bits and another 0-bit. True and false are represented as '1'B and '0'B respectively.

The letter B is the crucial element that distinguishes a BIT constant from a CHARACTER constant. For example, the constant '10110' is a CHARACTER constant, containing the five characters shown. In terms of the internal representation of PL/I data, the CHARACTER constant would require five bytes of storage, while the BIT constant '10110'B would require only five bits of storage, and so would occupy less space than a byte. More complex forms of BIT constants are explained in the next section.

Use the DECLARE statement to specify that a variable is to have a BIT data type. For example, if your program contains the statement

```
DECLARE BST BIT(7);
```

then the variable BST has, as its value, a string containing seven bits.

For example, if your program executes the statement

```
BST = '0110111'B;
```

the seven bits in the string '0110111'B are assigned to the variable BST.

The variable BST has as its value precisely seven bits, no more and no less. If your program attempts to assign to BST a BIT string value containing other than seven bits, PL/I must modify the value before assigning it. As in the case of CHARACTER strings, PL/I pads a short string, and truncates a long string.

For example, if your program executes the statement

```
BST = '111000111000'B;
```

then PL/I truncates the string being assigned by throwing away or truncating the last five of the twelve bits in the value on the right-hand side. The result is that BST is assigned the value '1110001'B. On the other hand, if your program executes

```
BST = '101'B;
```

then PL/I pads the value being assigned by adding four 0-bits to it on the right-hand side, so that BST is assigned the value '1010000'B.

As with CHARACTER strings, you may specify that a variable is to be BIT VARYING, meaning that the length specified in the DECLARE statement is a maximum length, and that the variable can have any length less than or equal to that maximum. For example, if your program contains

```
DECLARE BSV BIT(6) VARYING;
```

then BSV is a variable whose value is a string of bits, and the string contains six or fewer bits. It is still true, of course, that if your program assigns to BSV a bit string longer than six bits, truncation takes place.

It was mentioned above that the BIT(1) data type in PL/I performs the same function that logical data types perform in other programming languages. Suppose your program contains the following statements:

```
DECLARE TEST BIT(1);
TEST = (X < 0);
..
IF TEST THEN CALL NEGATIVE;
```

In this program segment, the variable TEST is specified to be a logical variable. The assignment statement on the second line illustrates how a truth value can be assigned to TEST. If the variable X is negative, TEST is assigned the BIT(1) value '1'B, which stands for true. If X is 0 or greater than 0, TEST is assigned '0'B, which stands for false. The IF statement shows how your program may make a decision based on this variable TEST; if TEST has the value '1'B, control is transferred to the procedure NEGATIVE.

As in the case of CHARACTER strings, you may DECLARE a BIT string explicitly to be NONVARYING. For example, the variable BST described above could have been declared with the statement

```
DECLARE BST BIT(7) NONVARYING;
```

If you do not specify either VARYING or NONVARYING, PL/I supplies the default of NONVARYING.

Also, as in the case of CHARACTER strings, it is possible under certain circumstances for the length specified in the DECLARE statement to be an expression containing variables. These circumstances are described in a later section.

String Data Types: BIT Constants

The simplest form of a BIT constant is illustrated several times in the preceding section. Such a constant begins with an apostrophe, followed immediately by a string of zeros and ones, followed by another apostrophe and the letter B.

PL/I provides other rules for writing certain kinds of BIT constants more conveniently than in the simple form just described. These rules are illustrated in Table 5-7.

Table 5-7
BIT Constants

Line #	Bit Constant	Length (# of Bits)	Bits
1	'01101'B	5	01101
2	'1'B	1	1
3	''B	0	(None)
4	(5)'1'B	5	11111
5	(4)'101'B	12	101101101101
6	'476'B3	9	100111110
7	'1'B3	3	001
8	''B3	0	(None)
9	(2)'67'B3	12	110111110111
10	'9F3'B4	12	100111110011
11	(3)'F0'B4	24	111100001111000011110000
12	'23021'B2	10	1011001001
13	(3)'2'B2	6	101010
14	'01101'B1	5	01101

Lines 1 and 2 of this table illustrate the simple form of the BIT constant that has been described. Line 3 illustrates the null BIT string, a BIT string containing no bits whatsoever. This corresponds to the null string, which has been previously described in the section on CHARACTER constants.

Line 4 illustrates the use of repetition factors in BIT constants. As in the case of CHARACTER strings, the number 5 in parentheses tells PL/I that the string is to be repeated five times. For this reason, the constant (5)'1'B represents exactly the same value as the constant '11111'B, and both of these constants have the same data type, BIT(5). Line 5 illustrates this concept further, where the repetition factor 4 applies to all three of the bits, 101.

Octal Notation: For your convenience, PL/I also permits you to write BIT constants using digits in the octal (base 8) number system. Lines 6 and 7 illustrate this. In the example '476'B3, the B3 signals to PL/I that the characters between the apostrophes are to be interpreted as octal digits, and that each of these digits is to be transmitted into three bits to create the final BIT constant. Therefore, the 4 becomes 100, the 7 becomes 111, and the 6 becomes 110, using the standard convention for translating octal digits to binary digits. That is why PL/I considers this constant to be exactly the same as the constant '100111110'B. Both constants have the same data type BIT(9), and both constants have the same value in terms of the string of bits. Line 7 also illustrates the use of B3. Here, the octal digit 1 is translated into the three bits 001, and so the constant '1'B3 is the same as the constant '001'B.

Lines 8 and 9 illustrate the use of B3 constants with other features. In line 8 the constant ''B3 is another way of writing the null BIT string, which was described before as ''B. In line 9, the repetition factor 2 applies to all the octal digits, so that the constant (2)'67'B3 is considered identical to the constant '6767'B3, which in turn is considered identical to the constant '110111110111'B. All three of these constants specify the same bit values, and all three have the data type BIT(12).

Other Number Bases: PL/I allows you to specify BIT constants in any of the common number bases: binary (base 2), quartal (base 4), octal (base 8), or hexadecimal (base 16). Specify your choice of the number base by means of the suffix B1, B2, B3, or B4, respectively. Note that B1 is the same as B. Each character that you specify between the apostrophes in a B1, B2, B3, or B4 constant represents, respectively, 1, 2, 3, or 4 bits in the final BIT string value. Table 5-8 shows which characters are legal for each of the four kinds of BIT constants, and shows how each of these characters is translated into a string of bits. Note that for hexadecimal (base 16) constants, which are specified by the suffix B4, the letters A through F are used to represent the hexadecimal digits for the decimal values 10 through 15.

Table 5-8

Characters Permitted in Bn Constants
and Corresponding Bit Values

Character	Corresponds to These Bits			
	B or B1	B2	B3	B4
0	0	00	000	0000
1	1	01	001	0001
2	Invalid	10	010	0010
3	Invalid	11	011	0011
4	Invalid	Invalid	100	0100
5	Invalid	Invalid	101	0101
6	Invalid	Invalid	110	0110
7	Invalid	Invalid	111	0111
8	Invalid	Invalid	Invalid	1000
9	Invalid	Invalid	Invalid	1001
A	Invalid	Invalid	Invalid	1010
B	Invalid	Invalid	Invalid	1011
C	Invalid	Invalid	Invalid	1100
D	Invalid	Invalid	Invalid	1101
E	Invalid	Invalid	Invalid	1110
F	Invalid	Invalid	Invalid	1111

Lines 10 through 14 in Table 5-7 illustrate some of these other kinds of BIT constants. In line 10, the constant '9F3'B4 is a hexadecimal BIT constant. Each of the characters between the apostrophes is translated into four bits, according to the usual hexadecimal to binary conversion rules as shown in Table 5-8. The character 9 translates into the bits 1001, the character F translates into the bits 1111, and the character 3 translates into the bits 0011, so that the equivalent constant is '100111110011'B. You may write this constant in either of these two formats; PL/I will consider them to have the same value as well as the same data type, BIT(12). In line 11, the repetition factor 3 applies to the two hexadecimal digits 10.

Lines 12 and 13 illustrate the B2 representation of a BIT constant. In line 12 each of the characters gets translated into two bits. In line 13, the repetition factor 3 is used.

Line 14 illustrates the alternate way, using B1, of writing the same constant as in line 1.

PICTURED DATA TYPES: INTRODUCTION

The CHARACTER data type, which has been described, is extremely powerful and flexible. CHARACTER variables can have, as their values, strings of any of the 256 characters supported by the Prime hardware. The use of such variables thus provides the programmer with the capability of manipulating any kind of character string in a wide variety of applications.

Unfortunately, this power carries with it the disadvantage of being a bit too general. In many applications, the programmer would like to declare CHARACTER variables in such a way that the strings that may be assigned to those variables are constrained to permit only certain characters in certain character positions. For example, a programmer may wish to specify that a certain CHARACTER variable may have as a value a string that can contain only letters as character values. Or, a programmer may wish to specify that a certain CHARACTER variable may contain only digits in its character value. In the latter case, the compiler should be able to interpret such a string as a numeric value as well as a string value.

Pictured data types provide these capabilities. The two kinds of pictured data types are pictured-string and pictured-numeric. They are described in the next sections.

PICTURED-STRING

A variable that is pictured-string is like a CHARACTER NONVARYING variable, except that only certain types of character string values may be assigned to it. In particular, it is possible for you to specify that certain character positions may contain only letters, and that other character positions may contain only digits. (PL/I will permit a blank in place of a letter or a digit.) You may also allow certain character positions to have any character.

Consider the following statement:

```
DECLARE CODE PICTURE 'AAAX99';
```

Notice that there are six characters between the two apostrophes following the keyword PICTURE. This declaration specifies that CODE is a CHARACTER variable containing exactly six characters, that the first three of these characters must be letters (represented by A), the next character may be any character (represented by X), and the last two characters must be digits (represented by 9). Thus, for example, the variable CODE may be assigned the value 'NUT-64', but may not have the value 'NT8-14' or the value 'NUT-X4'.

The simplest format for declaring a pictured string is as follows:

```
DECLARE variable PICTURE picture-specification;
```

where the picture specification has the format

```
'string-of-characters'
```

If such a declaration appears in your program, the specified variable will be CHARACTER NONVARYING, with a length equal to the number of characters between the apostrophes in the picture specification. The characters between the apostrophes may be any of the following:

- A specifies that the character in that position must be either a letter or a blank.
- X specifies that any character is permitted in that position.
- 9 specifies that the character in that position must be either a digit or a blank.

The picture specification must contain at least one A or X. If the picture specification contains only occurrences of the character 9, the variable is pictured-numeric, rather than pictured-string. Pictured-numeric variables are described in the next section.

If the same character appears several times consecutively in the picture specification, you may use a convenient shorthand notation. A parenthesized number appearing in a picture specification is interpreted as a repetition factor to be applied to the character immediately following. For example, the statement

```
DECLARE TYPE PICTURE 'X(7)AX(4)9';
```

is completely equivalent to the declaration

```
DECLARE TYPE PICTURE 'XAAAAAAAAAX9999';
```

In the first picture specification, the elements (7)A specify that the A is to be repeated seven times; this is done explicitly in the second picture specification. Similarly, the element (4)9 is replaced by 9999. An example of a CHARACTER string that is valid for assignment to TYPE is '*MAILBOX:8742'. Another example of a string that is valid for assignment to TYPE is '*SLOTbbb:b233', where b stands for a blank. This string is valid for assignment to TYPE because PL/I allows a blank

to be used in a character position where an A or a 9 was used in the picture specification.

In the declaration of a pictured variable, the keyword PICTURE may be abbreviated PIC.

PICTURED-NUMERIC

PL/I's pictured-numeric data type capability is a flexible method for constraining CHARACTER strings so that a legal value will be a character representation of a number in the format desired by the user. Furthermore, PL/I will interpret that variable as having an appropriate numeric value, if you desire. For example, you may specify that a variable is to contain precisely seven characters, and that the fifth of these seven characters must be a decimal point, while the others must all be digits. The string '8742.56' would be a legal value for that variable, but a string such as '-243492' would not be a valid value.

The pictured-numeric data type combines the functional advantages of the numeric data types with the functional advantages of the CHARACTER data type. The format for declaring a pictured-numeric variable is as follows:

```
DECLARE variable PICTURE picture-specification;
```

where PICTURE may be abbreviated to PIC. This is the same format as for pictured-string, described in the previous section. The difference is that the picture specification contains different characters for pictured-numeric, and these characters are interpreted quite differently.

Any pictured-numeric variable has two values, a numeric value and a CHARACTER value. When you use such a variable in a PL/I statement, PL/I uses whichever of the two values is appropriate for the context. For example, if the variable appears in an arithmetic expression, its numeric value is used. If you use PUT LIST to print out the value of the variable, its CHARACTER value is used. This means that you can manipulate the value of the variable numerically, and, when you print it out, get the formatting advantages of using a CHARACTER string.

Although the pictured-numeric data type has a number of advantages, it has one important disadvantage: numeric computations involving pictured-numeric variables execute much more slowly than the same computations involving ordinary numeric variables. This means that you should choose carefully which of your variables are to be pictured-numeric and which are to be numeric, based on how frequently each variable is to be used in arithmetic expressions. Of course, you may always use the assignment statement to assign to a pictured-numeric variable the value of a numeric variable, or vice versa.

The S and 9 Picture Specification Characters

Suppose your program contains the statement

```
DECLARE POINTS PICTURE 'S999';
```

This statement specifies that POINTS is to be pictured-numeric, meaning that it will have both a CHARACTER value and a numeric value.

CHARACTER Value: There are four characters in the picture specification above, and so the CHARACTER data type for POINTS is CHARACTER(4) NONVARYING. The first character in the CHARACTER value is always a sign (that is, + or -), because the first character in the picture specification is S. The next three characters in the CHARACTER value are always digits, since each of these positions has a 9 in the picture specification.

Numeric Value: The picture specification specifies that the variable is to contain three decimal digits. For this reason, the data type of the numeric value of POINTS is FIXED DECIMAL(3).

To understand how this variable is used, suppose your program executes the following statements:

```
POINTS = 23;  
POINTS = POINTS+1;  
PUT LIST(POINTS);
```

The first assignment statement assigns the numeric value 23 to POINTS. When PL/I executes this statement, it edits the numeric value into a character string that fits the specification given in your PICTURE declaration. This means that POINTS is assigned the value '+023'.

The second assignment statement increases the numeric value of POINTS by 1. In evaluating the expression POINTS + 1, PL/I uses the numeric value (23) of points, adds 1 to it, and then completes the assignment statement by editing the result 24 into a string that conforms to the picture specification. The result is that POINTS is assigned the value '+024'. When the PUT statement is executed, PL/I uses the string value of POINTS, and prints the following output value:

```
+024
```

Thus, PL/I moves freely back and forth between the numeric and string value of POINTS.

Table 5-9 illustrates the effect of assigning various values to pictured-numeric variables. Each line of this table shows what happens when you assign a certain value to a variable with a specified picture specification. The table shows what the resulting numeric and string values of the PICTURE variable will be.

Note

Internal representation of the numeric value for PICTURE is decimal data type.

Lines 1 through 6 illustrate what happens when various values are assigned to POINTS, the variable we have been discussing with the picture specification 'S999'. Lines 1 and 2 illustrate the basic feature that the string value of the variable contains a plus or minus sign, followed by three decimal digits. Line 3 of the table illustrates how truncation works, just as for any FIXED decimal variable. The fractional portion, point 8, of the value assigned is simply thrown away, with the result that the variable has a numeric value of 468 and a string value of '+468'. Line 4 illustrates what happens when you assign a value containing too many digits to POINTS. The result is a SIZE error, and the value of POINTS is undefined.

Lines 5 and 6 illustrate that you may also assign a string value to a pictured-numeric variable, provided that the string value corresponds to the requirements of the picture specification. In line 5, the value '+764' is assigned to POINTS, with the result that POINT has a numeric value of 764 and a string value of '+764'. Line 6 illustrates an invalid assignment of this type, where the assigned value '+7.3' does not conform to the picture specification, since the decimal point appearing in the third character position is not a digit.

Line 7 illustrates an alternate way of writing a picture specification when the same character is repeated several times. In this example, PL/I interprets the statement

```
DECLARE MILES 'S(7)9';
```

exactly the same as the statement

```
DECLARE MILES 'S9999999';
```

That is, the group (7)9 is interpreted as meaning that the 9 is to be repeated seven times.

Table 5-9

Assigning Values to
Numeric PICTURE Variables

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	23	'S999'	+023	'+023'	
2	-23	'S999'	-023	'-023'	
3	468.8	'S999'	+468	'+468'	Truncation
4	1874	'S999'	Invalid		SIZE error -- too many digits
5	'+764'	'S999'	+764	'+764'	CHARACTER assignment
6	'+7.3'	'S999'	Invalid		CHAR value does not conform to picture specification
7	-4.3E5	'S(7)9'	-430000	'-0430000'	Repetition factor
8	746	'999S'	+746	'746+'	Sign at end
9	-53	'999S'	-053	'053-'	
10	79942	'99999'	+79942	'79942'	No sign
11	874	'99999'	+00874	'00874'	
12	-874	'99999'	Invalid		SIZE error -- no character position for sign

Lines 8 and 9 illustrate that the plus or minus sign may appear at the end of the string value, as well as at the beginning. If you assign the value 746 to a pictured-numeric variable with picture specification '9999S', the variable has the numeric value 746 and a string value of '0746+'. If you assign -53 to the same variable, the string value is '0053-'.

Lines 10 through 12 of the table illustrate a picture specification that contains no sign at all. As shown in line 11, if you assign the value 874 to a variable with a picture specification of '99999', the variable has a numeric value of 874 and a string value of '00874'.

As illustrated in line 12, it is invalid to assign a negative value to such a variable, since there is no sign position in the picture specification. The result of such an assignment is a SIZE error.

Other Sign Symbols: -, +, CR, DB

It was explained above that an S in the picture specification is replaced by a plus or minus sign in the string value of the picture variable. PL/I provides a number of other ways to represent a sign in the string value. Some of these are as follows:

- If the picture specification contains a minus sign, PL/I inserts into the string value of the variable either a blank, if the numeric value is positive or zero, or a minus sign if the numeric value is negative. This form of string is, for many people, the most natural representation of the sign since a plus sign never appears, and a minus sign appears for negative numbers, which is the usual convention.
- If the picture specification contains a plus sign, PL/I inserts into the corresponding position in the string value of the variable a plus sign if the numeric value is zero or positive, and a blank if the numeric value is negative.
- In commercial applications involving billing, a negative value is considered a credit. If you use CR in a picture specification, PL/I inserts CR into the string value of the variable if the numeric value is negative. It inserts two blanks if the numeric value is zero or positive. CR may be used only at the end of the picture specification.
- In some commercial accounting applications, a negative value is considered to be a debit. If your picture specification contains DB, PL/I puts DB into the corresponding characters in the string value of the variable, provided that the numeric value is negative. If the numeric value is positive or zero, PL/I uses two blanks. Notice that, even though the terms credit and debit have opposite meanings in accounting, CR and DB work the same way in PL/I picture specifications.

PL/I Reference Guide

Table 5-10 illustrates the use of these picture specification symbols. Lines 1 through 6 illustrate the contrasting uses of S, +, and -. Lines 7 through 12 illustrate similar examples, with the sign appearing at the end of the picture specification. Lines 13 through 16 illustrate the use of CR and DB.

All of the examples in Table 5-10 have picture specifications with three positions for digits. As a result, the data type of the numeric value of the variable in each case is FIXED DECIMAL(3). The data type for the string value of the variable is CHARACTER(4) for lines 1 through 12 and CHARACTER(5) for lines 13 through 16.

DATA TYPES AND DATA ATTRIBUTES

Table 5-10

Assigning Values to Signed PICTURE Variables

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	23	'S999'	+023	'+023'	Sign Positive
2	23	'-999'	+023	'b023'	Sign Positive
3	23	'+999'	+023	'+023'	Sign Positive
4	-23	'S999'	-023	'-023'	Sign Negative
5	-23	'-999'	-023	'-023'	Sign Negative
6	-23	'+999'	-023	'b023'	Sign Negative
7	486	'999S'	+486	'486+'	Sign Positive at End
8	486	'999-'	+486	'486b'	Sign Positive at End
9	486	'999+'	+486	'486+'	Sign Positive at End
10	-486	'999S'	-486	'486-'	Sign Negative at End
11	-486	'999-'	-486	'486-'	Sign Negative at End
12	-486	'999+'	-486	'486b'	Sign Negative at End
13	18	'999CR'	+018	'018bb'	Sign Positive at End
14	18	'999DB'	+018	'018bb'	Sign Positive at End
15	-18	'999CR'	-018	'018CR'	Sign Negative at End
16	-18	'999DB'	-018	'018DB'	Sign Negative at End

Insertion Character Symbols: . , / B

When one of these four symbols appears in a picture specification, PL/I inserts the corresponding character into the string value of the variable. In the case of the symbol B in the picture specification, PL/I inserts a blank into the string value of the variable.

Table 5-11 illustrates the use of these four insertion character symbols. As this table illustrates, when a numeric value is assigned to a variable with a picture specification containing one of the insertion character symbols, PL/I inserts the appropriate character into the string value of the variable.

Table 5-11
Insertion Character Symbols

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of Variable String	Comment
1	8742935	'99,999,999'	+08742935	'08,742,935'	
2	780213	'99/99/99'	+780213	'78/02/13'	
3	9643	'S999.99'	+09643	'+096.43'	
4	-4	'99.99CR'	-0004	'00.04CR'	
5	783	'9,999.99'	+000783	'0,007.83'	
6	2743	'-99B9B9'	+2743	'b27b4b3'	
7	6.78	'S999.99'	+00006	'+000.06'	Truncation

It is important to note that the appearance of a decimal point in the picture specification does not mean that the numeric value of the PICTURE variable can contain a fraction. This is illustrated in lines 3, 5, and 7 in the table. In particular, line 7 illustrates that when the value 6.78 is assigned to a variable with picture specification 'S999.99', the numeric value is truncated with the result that the variable has the numeric value 6 and the string value '+000.06'.

In each of the examples in Table 5-11, the string value of the variable has a data type CHARACTER NONVARYING, with a string length equal to the number of characters in the picture specification. The data type for the numeric value of the variable is FIXED DECIMAL, with a precision (number of digits) equal to the number of occurrences of 9 in the picture specification. For example, a variable declared to have this picture specification '99,999,999', as illustrated in line 1 of the table, has a string data type of CHARACTER(10), and a numeric data type of FIXED DECIMAL(8).

Numeric Scale Factor Symbols: V and F(n)

All the numeric picture specifications we have discussed so far could be used to represent only integer values. In other words, in all cases, the data type of the numeric value of the PICTURE variable was FIXED DECIMAL with a scale factor of 0. This section deals with ways of introducing a nonzero scale factor into the picture specification.

The easiest way to allow noninteger numeric values is to use V plus a period (V.) as a single element to specify where the decimal point occurs. This usage is illustrated in lines 1 through 6 of Table 5-12. For example, line 1 shows the result of assigning the value 4 to a variable with picture specification '-9V.99'. The symbol 9 appears three times, meaning that the numeric value of the variable contains three decimal digits. Since two occurrences of 9 occur after V plus a period, two of the digits in the numeric value of the variable appear after the decimal point. The result is that when 4 is assigned to the variable, the resulting numeric value is 4.00, and the resulting string value is 'b4.00'. If -8.6 is assigned to the same variable, the resulting numeric is -8.60, and the resulting string value is '-8.60'. If 7.894 is assigned to the same variable, then, since the variable can only accommodate two digits after the decimal point, truncation takes place, the resulting numeric value is 7.89, and the string value is 'b7.89'. These results are illustrated in lines 2 and 3 of Table 5-12.

The symbol V is unusual because it appears in the picture specification but does not correspond to any character in the string representation of the value of the variable. That is, even though the picture specification '-9V.99' contains six characters, the data type for the string value of the variable is CHARACTER(5). Each of the characters in the picture specification contributes one character to the string representation of the variable except for V, which is ignored in the string representation. On the other hand, the V does affect the data type of the numeric value of the variable, since the scale factor of the data type is determined by the number of occurrences of 9 that follow the V. Thus, the picture specification '-9V.99' has a corresponding numeric type of FIXED DECIMAL(3,2).

Lines 4 and 5 illustrate another picture specification using V followed by a period. In this case, each occurrence of 9 follows the V, meaning that all the digits in the value of the variable follow the decimal point. The string data type of the value of the variable is CHARACTER(5), and the numeric data type is FIXED DECIMAL(3,3). As illustrated in line 5, attempting to assign a value greater than one to such a variable yields a SIZE error.

Table 5-12

Assigning Values to PICTURE
Variables with Scale Factors

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	4	'-9V.99'	+4.00	'b4.00	
2	-8.6	'-9V.99'	-8.60	'-8.60'	
3	7.894	'-9V.99'	+7.89	'b7.89'	Truncation
4	.2	'SV.999'	+.200	'+.200'	
5	4	'SV.999'	Invalid		SIZE error
6	12.3	'99V.99'	+12.30	'12.30'	
7	12.3	'99.99'	+0012	'00.12'	Truncation
8	12.3	'99V99'	+12.30	'1230'	
9	83	'99V99'	+83.00	'8300'	
10	.25	'99V99'	+00.25	'0025'	
11	493	'99V99'	Invalid		SIZE error
12	.3	'V.9999'	+.3000	' .3000'	
13	.3	' .9999'	+.0000	' .0000'	Truncation
14	.3	'V9999'	+.3000	'3000'	
15	.029	'V9999'	+.0290	'0290'	
16	3	'V9999'	Invalid		SIZE error
17	487000	'999F(3)'	+487F3	'487'	
18	29763	'999F(3)'	+29F3	'029'	Truncation
19	12.3	'99.99F(-2)'	+12.30	'12.30'	Same as lines 6-8
20	12.3	'99.99'	+0012	'00.12'	
21	12.3	'9999F(-2)'	+12.30	'1230'	
22	.00003	'99F(-6)'	+30F-6	'30'	
23	.0000297	'99F(-6)'	+29F-6	'29'	Truncation
24	43.782	'99V99F(1)'	+04.37	'0437'	Truncation

It is not necessary to use the symbols V and period together. Earlier, you saw that the period is an insertion character that PL/I copies to the string value of the variable without affecting the numeric value of the variable. On the other hand, the symbol V does not correspond to any character in the string value of the PICTURE variable, but since PL/I uses the V to determine the scale factor, this symbol does affect the numeric value of the variable. Therefore, the period affects only the string value of the PICTURE variable, and V affects only the numeric value. When they are used together (V.), they affect both the string and numeric values and they do so in a consistent manner, since the position of the decimal point in the string value corresponds to the scale factor in the numeric value.

Lines 6, 7, and 8 of Table 5-12 illustrate what happens when the period and the V are used separately. In line 6, the value 12.3 is assigned to a variable with picture specification '99V.99'. Here the V and the period are used together, and, as shown, the resulting numeric value is

12.3 and the resulting string value '12.30'. The string data type is CHARACTER(5), and the numeric data type is FIXED DECIMAL(4,2).

In line 7, there is only a period in the picture specification. Since there is no V, the scale factor of the numeric value is 0, and so the numeric value can only have an integer value. For this reason, when the value 12.3 is assigned, truncation occurs, and the resulting numeric value is 12, corresponding with the numeric data type of FIXED DECIMAL(4). In the string value of the variable, the numeric value of 12 is represented by means of the digits 0012, but the period in the picture specification is simply copied over to the string value of the variable, resulting in a string value of '00.12'.

Line 8 has the first example of a picture specification containing a V with no period. The data type of the numeric value of the variable is FIXED DECIMAL(4,2). This example is identical to the one in line 6, where the period appears, with the exception that there is no decimal point appearing in the string value of the variable. However, the numeric value is the same in both cases, and the digits appearing in the string value are the same. Lines 10 and 11 provide two additional examples of assignment to the same picture specification as in line 9. Lines 12 through 16 of the table provide a further illustration of the concepts just described. In this example, all of the digits appear after the decimal point.

Lines 17 and 18 illustrate a new kind of picture specification. The appearance of F(3) at the end of the picture specification indicates that the numeric value of the variable will have three zeros following the digits as they appear in the string value. Thus, for example, if the value 487000 is assigned to a variable with the picture specification '999F(3)', the resulting numeric value is 487000 (which could also be written as 487F3), and the string value is '487'. For this picture specification, the numeric data type is FIXED DECIMAL(3,-3), and the string data type is CHARACTER(3). In line 18, the value 29763 is assigned to a variable with the same picture specification. Truncation takes place, and so the resulting numeric value is 29000, and the resulting string value is '029'.

Like V, when F(n) appears in a picture specification it determines the scale factor of the numeric value of the variable. The rule is as follows: an increase of one in the value of n in the F(n) is equivalent to moving the V to the left one position with respect to the occurrences of 9 in the picture specification. Lines 19, 20, and 21 illustrate this. The picture specifications on these three lines are entirely equivalent to the picture specification in lines 6, 7, and 8. The appearance of F(-2) is equivalent to inserting V to the left of two occurrences of 9 in the picture specification.

In lines 22 and 23, the implied decimal point is six positions to the left of the rightmost digit appearing in the string value of the variable. In line 23, truncation takes place.

It is even possible to use both V and F in the same picture specification, resulting in their effects being combined. The picture

specification '99V99F(1)' is equivalent to the picture specification '999V9'.

The following is a summary of the effects of V and F(n) on the numeric and string data types of the variable. The string data type is CHARACTER NONVARYING, with the length equal to the number of characters in the picture specification, not including the V or the F(n). The numeric data type is FIXED DECIMAL(p,q), where p equals the number of occurrences of 9 in the picture specification, and q equals $m-n$, where m is the number of occurrences of 9 following the symbol V, and n is the integer appearing along with F.

Suppressing Zeros: Z, *, and Y

One of the main purposes of PICTURE variables is to provide a method of storing numeric values in a string form that is suitable for attractive or functional output. One important feature of attractive output is the suppression of leading zeros. For example, in printing 45, most users would consider the output 0045 unattractive, because of the two leading zeros.

In a string specification, the symbol Z has the same meaning as 9, except that, in the string value of the variable, PL/I will substitute a blank for the Z if the digit that would have been substituted was a leading zero.

In Table 5-13, lines 1 through 5 illustrate what happens when various values are assigned to a PICTURE variable with the picture specification 'ZZ99'. The data type of the numeric value of the variable is FIXED DECIMAL(4), and the data type of the string value of the variable is CHARACTER(4), just as if the picture specification were '9999'. In fact, the Z is entirely equivalent to 9 in determining the data types of the string and numeric values of the PICTURE variable. The difference is that PL/I does not always substitute a digit for a Z as it does for a 9.

DATA TYPES AND DATA ATTRIBUTES

Table 5-13
Zero Suppression With PICTURE Variables

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	8002	'ZZ99'	+8002	'8002'	No suppression
2	453	'ZZ99'	+0453	'b453'	One digit suppressed
3	62	'ZZ99'	+0062	'bb62'	Two digits suppressed
4	4	'ZZ99'	+0004	'bb04'	
5	0	'ZZ99'	+0000	'bb00'	
6	400	'ZZZ'	+400	'400'	No suppression
7	5	'ZZZ'	+005	'bb5'	Two digits suppressed
8	0	'ZZZ'	+000	'bbb'	All digits suppressed
9	-25	'SZZ9'	-025	'-b25'	
10	0	'ZZ9S'	+000	'bb0+'	
11	4279365	'Z,ZZZ,ZZ9'	+4279365	'4,279,365'	No suppression
12	279365	'Z,ZZZ,ZZ9'	+0279365	'bb279,365'	Digit and comma suppressed
13	143	'Z,ZZZ,ZZ9'	+0000143	'bbbbbb143'	Both commas suppressed
14	2	'ZZ/Z9/99'	+000002	'bbbb0/02'	Slash suppressed
15	8002	'**99'	+8002	'8002'	Similar to lines 1-4
16	453	'**99'	+0453	'*453'	Similar to lines 1-4
17	62	'**99'	+0062	'**62'	Similar to lines 1-4

Table 5-13 (continued)
Zero Suppression With PICTURE Variables

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
18	4	'**99'	+0004	'**04'	Similar to lines 1-4
19	143	'*,**,*9'	+0000143	'*****143'	Similar to lines 13-14
20	2	'**/*9/99'	+000002	'****0/02'	
21	10203	'YYYYY'	+10203	'1b2b3'	
22	100000	'YY/Y9/Y9'	+100000	'1b/b0/b0'	

As line 1 of Table 5-13 shows, when the value 8002 is assigned to such a variable, the resulting string value is '8002', since there are no leading zeros. However, as line 2 shows, when 453 is assigned to the same variable, the resulting string value is 'b453'. The leading zero is replaced with a blank. When the value 62 is assigned, two leading zeros are replaced with blanks, as illustrated in line 3. On the other hand, when the value 4 is assigned to the variable as illustrated in line 4, only two of the three leading zeros are suppressed, since a 9 appears in the third position of the picture specification. Furthermore, as line 5 shows, when the value 0 is assigned, the resulting string value is 'bb', since the two occurrences of 9 at the end of the picture specification require actual digit substitution.

Lines 6, 7, and 8 illustrate assignment to a variable with a picture specification consisting entirely of the character Z. For a variable with picture specification 'ZZZ', the numeric value has data type FIXED DECIMAL(3), and the string value has data type CHARACTER(3). When the value 400 is assigned to this variable, the resulting string value is '400', since there are no leading zeros. When the value 5 is assigned, the resulting string value is 'bb5', since there are two leading zeros. When the value zero is assigned, all three digits are suppressed, and the resulting string value contains three blanks.

The use of the various sign symbols is not affected at all by the suppression of leading zeros. Lines 9 and 10 of Table 5-13 illustrate this.

An important feature of the suppression of leading zeros is that any insertion characters encountered along the way during the editing process are replaced with blanks if the preceding digits are replaced with blanks. This feature is illustrated in lines 11 through 14. These examples illustrate the suppression of the insertion characters comma (,) and slash (/) when leading zeros are suppressed.

In certain commercial applications, such as the printing of checks, it is desirable that leading zeros be replaced by some printing character, rather than by a blank. The character most commonly used is the asterisk (*), and PL/I provides this capability. When the symbol * appears in a picture specification, it is treated precisely the same as a Z, except that, when a digit is suppressed, it is replaced with an asterisk rather than a blank. These features are illustrated by lines 13 through 20 of Table 5-13. These examples are identical to the examples in lines 1 through 4 and 13 through 14, except that an asterisk is used instead of Z.

The last zero-suppression symbol is Y. When this symbol appears in a picture specification, the corresponding string value has any zero replaced with a blank, whether the digit is leading or not. This is illustrated in lines 21 and 22 of Table 5-13.

The symbols Z and * may not be intermixed with one another, nor may they be intermixed with 9 or Y. Also, Z and * may not both appear in the same picture specification. Furthermore, no 9 or Y may appear to the left of either a Z or an * in the picture specification. The symbols 9 and Y may be intermixed as desired.

In the preceding section, the data types of the string and numeric values of a pictured variable have been defined in terms of the number of occurrences of 9. Now those rules must be amended to count occurrences of the symbols 9, Z, *, and Y.

Suppressing Leading Zeros in Picture Specifications with V

When V appears in the same picture specification as either Z or *, some special rules apply. These rules are as follows:

- If either Z or * appears in the picture specification to the right of V, all digit positions in the picture specification must be represented by Z or *, respectively.
- When your program assigns a numeric value to a picture specification all of whose digit positions contain either Z or *, then, in creating the string value of the variable, PL/I does not suppress any digit or insertion character appearing to the right of a V, even a leading zero, unless the numeric value is zero. In the latter case, all digits and insertion characters are suppressed.

Table 5-14 illustrates these rules. In line 2, the insertion character period is not suppressed, as it would be if the picture specification contained no V. As illustrated in line 3, the picture specification 'ZZV.Z9' is an illegal picture specification, since a Z appears to the right of a V, but there is still a 9 in the picture specification.

The case where all digit positions are occupied by Z, with two of those digit positions following a V, is illustrated in lines 4 through 7. The only case where digits after the V are suppressed is illustrated in line 7, where the numeric value is 0, and all digit positions, as well as the insertion character period, are suppressed.

When the insertion character period appears to the left of V, as illustrated in lines 8 and 9, it can be suppressed even if digits to the right of V are not suppressed. In line 9, the resulting string value 'bbb42' could cause confusion to somebody reading the output from your program. That is why V followed by a period (V.) is considered preferable in most circumstances to period followed by a V (.V).

Lines 10 through 13 of the table are similar to lines 4 through 7 except that the insertion character period has been omitted. This serves to emphasize the point that a period is inserted into the string value of the variable, and carries no meaning as regards the numeric value of the variable, which is the function of V.

Lines 14 through 20 of the table are the same as lines 1 through 7 of the table, except that the zero suppression character * replaces Z. All other remarks are the same.

Drifting Signs: S, +, and -

In the string value of a PICTURE variable, in many cases you would prefer that there be no blanks between the sign and the first nonzero digit. The use of drifting signs solves this problem. When PL/I edits a numeric value into the string value, it lets the sign drift to the right so that it appears just before the first nonzero digit.

Table 5-15 illustrates the use of drifting signs. In the first two lines, the symbol S is a static sign, so called because it can appear in only one position, the leftmost position of the string value of the variable. In line 2, where leading zeros have been suppressed, there are blanks between the minus sign and the 2. Lines 3 through 9 illustrate the various results when different values are assigned to a variable whose picture specification contains S as a drifting sign. In each case, the sign appears in the position immediately preceding the first significant digit.

DATA TYPES AND DATA ATTRIBUTES

Table 5-14
Assigning Values to PICTUREs With V

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	8.42	'ZZV.99'	+08.42	'b8.42'	
2	.42	'ZZV.99'	+00.42	'bb.42'	
3		'ZZV.Z9'			Illegal picture specification
4	8.42	'ZZV.ZZ'	+08.42	'bb.42'	
5	.42	'ZZV.ZZ'	+00.42	'bb.42'	
6	.02	'ZZV.ZZ'	+00.02	'bb.02'	0 after V not suppressed
7	0	'ZZV.ZZ'	+00.00	'bbbb'	All digits and period suppressed
8	8.42	'ZZ.VZZ'	+08.42	'b8.42'	
9	.42	'ZZ.VZZ'	+00.42	'bbb42'	
10	8.42	'ZZVZZ'	+08.42	'b842'	Similar to lines 4-7
11	.42	'ZZVZZ'	+00.42	'bb42'	Similar to lines 4-7
12	.02	'ZZVZZ'	+00.02	'bb02'	Similar to lines 4-7
13	0	'ZZVZZ'	+00.00	'bbbb'	Similar to lines 4-7
14	8.42	'**V.99'	+08.42	'*8.42'	
15	.42	'**V.99'	+00.42	'**.42'	
16		'**V.*9'			Illegal picture specification
17	8.42	'**V.**'	+08.42	'*8.42'	
18	.42	'**V.**'	+00.42	'**.42'	
19	.02	'**V.**'	+00.02	'**.02'	0 after V not suppressed
20	0	'**V.**'	+00.00	'*****'	All digits and period suppressed

Table 5-15
Assigning Values to PICTUREs With Drifting Signs

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	-25	'S99999'	-00025	'-00025'	Static sign
2	-25	'SZZZZ9'	-00025	'-bbb25'	Static sign
3	-25	'SSSSS9'	-00025	'bbb-25'	Drifting sign
4	0	'SSSSS9'	+00000	'bbbb+0'	Drifting sign
5	25	'SSSSS9'	+00025	'bbb+25'	Drifting sign
6	3875	'SSSSS9'	+03875	'b+3875'	Drifting sign
7	42765	'SSSSS9'	+42765	'+42765'	Drifting sign
8	-42765	'SSSSS9'	-42765	'-42765'	Drifting sign
9	723463	'SSSSS9'	Invalid		SIZE error
10	25	'SSSSSS'	+00025	'bbb+25'	Drifting sign, all positions
11	0	'SSSSSS'	+00000	'bbbbbb'	
12	4876	'SS,SS9'	+4876	'+4,876'	Comma inserted
13	876	'SS,SS9'	+0876	'bb+876'	Comma replaced with sign
14	76	'SS,SS9'	+0076'	'bbb+76'	Comma replaced with blank
15	.03	'SS9V.99'	+00.03	'b+0.03'	
16	.03	'SSSV.99'	+00.03	'bb+.03'	
17	.03	'SSSV.SS'	+00.03	'bb+.03'	
18	0	'SSSV.SS'	+00.00'	'bbbbbb'	
19		'SSSV.S9'			Illegal picture specification

Table 5-15 (continued)
Assigning Values to PICTUREs With Drifting Signs

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
20	25	'-----9'	+00025	'bbbb25'	Other drifting signs
21	-25	'-----9'	-00025	'bbb-25'	Other drifting signs
22	25	'+++++9'	+00025	'bbb+25'	Other drifting signs
23	-25	'+++++9'	-00025	'bbbb25'	Other drifting signs

Specify a drifting sign in a picture specification by using the sign two or more times in the specification. If you use a drifting sign, all occurrences of the drifting sign must appear to the left of any of the digit symbols 9, Z or Y.

For computing the data type of the string value of the PICTURE variable, each occurrence of the drifting sign symbol contributes one character to the string value. Thus, a variable with a picture specification of 'SSSSS9' would be CHARACTER(6); that is, all six characters in the picture specification correspond to characters in the string value. On the other hand, for computing the data type of the numeric value of the picture variable, all but one of the occurrences of the drifting sign symbol contribute to the precision. That is, if there are n occurrences of a drifting symbol in the picture specification, only $(n - 1)$ of these contribute to the precision of the numeric value of the PICTURE variable. For this reason, the picture specification 'SSSSS9' has a numeric data type of FIXED DECIMAL(5). The precision of 5 comes from the four digit positions contributed by the five occurrences of S, plus one from the occurrence of 9.

It is possible for a picture specification to contain no digit symbols other than the drifting sign symbols as illustrated in lines 10 and 11. When zero is assigned to a variable with such a picture specification, the string value contains all blanks. Notice that the data type of the numeric value of a variable with picture specification 'SSSSS' is FIXED DECIMAL(5). The precision of 5 comes from the six occurrences of S.

When an insertion character, such as a comma, appears in the picture specification either in the midst of, or immediately following, a string of drifting sign symbols, then PL/I includes the insertion

character, in a sense, as part of the drifting field. The sense is that the insertion character can become any of the following in the final edited string value of the PICTURE variable:

- If a nonzero digit appears before the insertion character, the insertion character is left unchanged.
- If the first nonzero digit appears immediately after the insertion character, the insertion character is replaced by a sign, just as if the insertion character were part of the drifting sign field.
- If the first nonzero digit appears after that point, the insertion character is replaced with a blank, as usual.

These rules are illustrated in lines 12 through 14 of the table. Notice, in particular, line 13, where the insertion character comma was replaced with the plus sign.

A drifting sign interacts with V very much as Z does. If a drifting sign appears to the right of a V, it must appear in all positions. No digits to the right of V are suppressed in the string value of the PICTURE variable unless the numeric value of the variable is 0, in which case the entire field is suppressed. These rules are illustrated in lines 15 through 19 of the table.

This discussion regarding the sign symbol S applies equally to the symbols + and -. As usual, the minus sign is replaced by a blank if the numeric value is greater than or equal to 0, and the plus sign is replaced with a blank if the numeric value is less than 0.

Static and Drifting \$

If you wish a currency symbol to appear in the string value of the PICTURE variable, use a \$ in the picture specification. If there is only one \$ in the picture specification, it is treated as a static symbol, and PL/I simply copies it when editing the string value of the picture variable. This is illustrated in line 1 of Table 5-16.

Table 5-16
Static and Drifting Dollar Sign

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	25	'\$ZZZZ9'	+00025	'\$bbb25'	Static \$
2	25	'\$\$\$\$\$9'	+00025	'bbb\$25'	Drifting \$
3	87425	'\$\$\$\$\$9'	+87425	'\$87425'	Drifting \$
4	2	'\$\$\$\$\$9'	+00002'	'bbbb\$2'	Drifting \$
5	-23	'\$\$\$\$\$9'	Invalid		SIZE error -- no sign in picture specification
6	8742.63	'\$\$,\$\$V.99DB'	+8742.63	'\$8,742.63DB'	
7	-834.92	'\$\$,\$\$V.99DB'	+0834.92	'bb\$834.92bb'	
8	6.23	'\$\$,\$\$V.99CR'	+0006.23	'bbbb\$6.23bb'	
9	.02	'\$\$,\$\$V.99CR'	+0000.02	'bbbb\$.02bb'	
10	-7.63	'\$\$,\$\$V.99CR'	-0007.63	'bbb\$7.63CR'	

If there are two or more occurrences of \$ in the picture specification, it is treated as a drifting character, and the rules for it are the same as for the drifting signs. Lines 2 through 10 of the same table illustrate these features. Notice that the currency symbol drifts to the right until it appears before the first digit in the string.

Since it is illegal to have a drifting sign in the same picture specification as a drifting currency symbol, use either CR or DB to indicate the sign of the numeric value. The use of CR is illustrated in lines 6 through 10.

Overpunched Sign Symbols: T, I, and R

In the early days of punched-card usage with the BCD (binary-coded decimal) character set, it was the convention that a signed number could be punched by overpunching the last digit of the number with a plus sign or minus sign. When such overpunching occurred, the result was simply a new character. For example, when the digit 1 is overpunched with a plus sign (12-punch), the result is the character A. If 1 is overpunched with a minus sign (11-punch), the result is the character J. Table 5-17 provides a complete list of these overpunched codes.

Table 5-17
Digits With Overpunched Signs

Digit	Overpunched Digit	
	With +	With -
0	{	}
1	A	J
2	B	K
3	C	L
4	D	M
5	E	N
6	F	O
7	G	P
8	H	Q
9	I	R

If you use the symbol T in the picture specification for a variable, you are indicating that the sign of the number is to be specified by one of these overpunched characters. The symbol T is handled exactly like 9, except that the corresponding character in the string value of the PICTURE variable will contain an overpunched digit rather than a simple digit. This is illustrated in lines 1 through 6 of Table 5-18. Notice that, as illustrated in lines 5 and 6, the symbol T need not appear in the last position of the picture specification.

Note

Because of a Prime hardware limitation, only negative sign overpunch is available in Prime PL/I.

Table 5-18
PICTURE Variables With Overpunch

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	25	'999T'	+0025	'002E'	+ or - overpunch
2	-25	'999T'	-0025	'002N'	+ or - overpunch
3	1282	'999T'	+1282	'128B'	+ or - overpunch
4	-1282	'999T'	-1282	'128K'	+ or - overpunch
5	1282	'99T9'	+1282	'12H2'	+ or - overpunch
6	-1282	'99T9'	-1282	'12Q2'	+ or - overpunch
7	1282	'999R'	+1282	'1282'	- overpunch only
8	-1282	'999R'	-1282	'128K'	- overpunch only
9	1282	'999I'	+1282	'128B'	+ overpunch only
10	-1282	'999I'	-1282	'1282'	+ overpunch only

The symbol R is like T except that only a negative numeric value causes an overpunch; if the numeric value is zero or positive, the R is replaced with an ordinary digit in the string value of the PICTURE variable. This is illustrated in lines 7 and 8 of the table. Similarly, lines 9 and 10 illustrate the use of I, which causes an overpunch only if the numeric value is zero or positive.

The following sign symbols have been discussed: S, +, -, CR, DB, \$, T, R, and I. Only one of these symbols may be used in any picture specification, with the exception of the use of S, +, or - as a drifting sign. In any case, two different sign symbols may not be used in the same field. If no sign symbols appear, the numeric value of the PICTURE variable must always be positive.

FLOAT Symbols: E and K

Up until now, all of the pictured-numeric variables that have been discussed have had numeric data types with a scale of FIXED. This section covers picture specifications that define variables with FLOAT numeric data types.

A FLOAT number can be represented by means of two fixed numbers, the mantissa and the characteristic (that is, the exponent). For this reason, a FLOAT picture specification consists of two FIXED picture specifications separated by the letter E.

Consider, for example, the picture specification 'SV.999ES99', which is illustrated in lines 1 through 5 of Table 5-19. This FLOAT picture specification contains two FIXED picture specifications. The first, 'SV.999', is for the mantissa, and the second, 'S99', is for the characteristic of the numeric value of the PICTURE variable. In creating the string value of the PICTURE variable, PL/I edits the mantissa using the first picture specification, copies the letter E, and then edits the characteristic using the second picture specification.

Table 5-19
Assigning Values to PICTUREs With FLOAT Symbols

Line #	Value Assigned	PICTURE Spec for Target Variable	Result: PICTURE Numeric	Values of PICTURE String	Comment
1	23	'SV.999ES99'	+.230E2	'+.230E+02'	
2	8.74	'SV.999ES99'	+.874E1	'+.874E+01'	
3	.0003	'SV.999ES99'	+.300E-3	'+.300E-03'	
4	-23	'SV.999ES99'	-.230E2	'-.230E+02'	
5	-.0003	'SV.999ES99'	-.300E-3	'-.300E-03'	
6	23	'S99V.9ES99'	+.230E2	'+23.0E+00'	
7	-.0003	'S99V.9ES99'	-.300E-3	'-30.0E-05'	
8	0	'S99V.9ES99'	+.000E0	'+00.0E+00'	
9	8.74	'SSSV.9ESS9'	+.874E1	'+87.4Eb-1'	
10	0	'SSSV.9ESS9'	+.000E0	'bb+.0Eb+0'	
11	87	'9E9'	+.8E2	'8E1'	
12	-87	'9E9'	Invalid		SIZE error -- no sign in mantissa
13	+.0003	'9E9'	Invalid		SIZE error -- no sign in characteristic
14	23	'SV.999KS99'	+.230E2	'+.230+02'	

The scale factor of the mantissa portion of the picture specification, as determined by the symbol V or F, determines the value of the characteristic, which is to be edited using the second portion of the picture specification. For example, in line 1 of the table, V appears before the first digit while, in line 6, V appears before the third digit. Although, in both of these lines, the numeric value of the variable is the same, the value appearing in the characteristic portion of the string value is different because the scale factor in the mantissa portion is different. (Compare lines 5 and 7 in the table.)

Use of drifting signs in the mantissa portion of a FLOAT picture specification usually does not make sense, because when PL/I creates the string value of the PICTURE variable, it uses as its first digit in the mantissa the leading significant digit in the numeric value. This means that there are usually no leading zeros to suppress (see line 9 of the table). The exception is when the numeric value is 0, as illustrated in line 10.

Notice that the mantissa and characteristic portions of the picture specifications have separate signs. It is possible to get a SIZE error in either portion, if there is no provision for a negative sign. This is illustrated in lines 11 through 13 where the picture specification '9E9' has no provision for a sign in either the mantissa or the characteristic.

The use of K is identical to the use of E, except that no E is inserted into the string value of the PICTURE variable. This is illustrated in line 14 of the table, which is identical to line 1, except that K appears instead of E in the picture specification, and no character whatsoever appears in the string value of the PICTURE variable to separate the mantissa and characteristic portions.

Some of the rules that have been stated in preceding sections must now be modified. All rules that stated "you can't do such and such in a picture specification" should now be rewritten as "you can't do such and such in a single field of a picture specification," where a field of a FLOAT picture specification is either the mantissa field or the characteristic field. For example, the rule that you may not have two different sign symbols in a picture specification must now be changed to state that you may not have two different sign symbols in a single field of a picture specification.

The rules for determining the data types for the numeric and string values of a PICTURE variable with a FLOAT picture specification are as follows: the numeric value has a data type of FLOAT DECIMAL, with a precision equal to the precision of the mantissa portion of the picture specification. (The FLOAT data type has no scale factor.) The string value has a data type of CHARACTER NONVARYING. The length of the string equals the sum of the length of the strings corresponding to the mantissa portion and characteristic portion of the picture specification. To this sum, add 1 if E is used in the picture specification; do not add 1 if K is used.

COMPLEX Pictures

If you wish the numeric value of a PICTURE variable to be COMPLEX, rather than REAL, specify the keyword COMPLEX in the declaration of the PICTURE variable. For example, suppose the statement

```
DECLARE CPP PICTURE 'S9V.99ES99' COMPLEX;
```

appears in your program. COMPLEX is combined with the data type derived from the picture specification, and the numeric value of CPP has a data type of FLOAT DECIMAL(3) COMPLEX. The string value of CPP contains two parts, one for the real part of the value, and one for the imaginary part. For this reason, the data type of the string value of CPP is CHARACTER(18) NONVARYING. For example, if your program executes the statement

```
CPP = 3-2I;
```

then the resulting numeric value of CPP is $+3.00E1-.200E1I$, and the string value of CPP is '+3.00E+00-2.00E+01'.

ARRAYS AND STRUCTURES

The preceding section dealt with the data types of individual PL/I data elements. In all the cases discussed, each variable represented a single data value, such as a number or a string. For this reason, these variables are called scalars. When a variable can stand for many individual data elements, it is an aggregate, and its value is called an aggregate value. The two basic kinds of aggregates are the array and the structure. In addition, it is possible to combine the basic aggregate types to get more complex aggregates, such as arrays of structures or structures of arrays.

One-dimensional Arrays

An array is an aggregate value all of whose individual data elements have the same data type. Consider, for example, the following statement:

```
DECLARE PRICES(50) FIXED DECIMAL(7,2);
```

DATA TYPES AND DATA ATTRIBUTES

If this statement appears in your program, it specifies that PRICES is an array containing 50 individual data elements or scalars. Each of these 50 individual data elements has the same data type, FIXED DECIMAL(7,2).

Figure 5-1 illustrates the array PRICES. When you wish to refer to all 50 scalar elements in PRICES at once, use the identifier PRICES. For example, a statement like

```
PRICES = 7;
```

says to set the entire array PRICES to 7, which PL/I interprets as meaning that each of the 50 scalar elements of PRICES is to be assigned the value 7. Similarly, the statement

```
PUT LIST(PRICES);
```

causes PL/I to print out each of the 50 data elements in the array PRICES.

To refer to each of the 50 scalars individually, use a subscript. For example, the statement

```
PRICES(49) = 15*PRICES(2);
```

is an assignment statement involving the individual elements of the array PRICES. As shown in the figure, PRICES(2) refers to the second of the scalar elements, while PRICES(49) refers to the 49th of the scalar elements. The subscripts are 2 and 49.

PRICES

PRICES (1)
PRICES (2)
PRICES (3)
PRICES (4)
•
•
•
PRICES (48)
PRICES (49)
PRICES (50)

Representation of the Array PRICES
Figure 5-1

Most of the real programming power of arrays comes from the fact that array subscripts may be variables or expressions. For example, something like

```
DO INDEX = 1 TO 50;  
  PRICES(INDEX) = INDEX;  
END;
```

is legal in PL/I, and permits you to do in a three-statement loop something that would require 50 statements if you used 50 individual variables rather than an array. Any PL/I expression may be used as a subscript, so that a statement like

```
PRICES(K + L) = PRICES(2*M);
```

is perfectly legal.

Array Bounds

The bounds of an array are the maximum and minimum values that a subscript used with the array can have. The lower bound of the array is the minimum value that a subscript can have, and the upper bound is the maximum value. In the array PRICES shown in Figure 5-1, the lower bound is 1, and the upper bound is 50.

Use a colon (:) in your array declaration to specify that the lower bound of the array should be other than 1. For example, the statement

```
DECLARE VECTOR(0:12) FLOAT;
```

specifies that VECTOR is an array of 13 elements, and that the lower bound of the subscript values is 0 and the upper bound is 12. By using this kind of declaration, you may make the lower bound any value you want, including a negative value.

Under certain circumstances, PL/I allows you, in your array declaration, to use as array bounds expressions containing variables. For example, the statement

```
DECLARE VALUES(N);
```

is legal, even though the upper bound, represented by N, is variable. The circumstances in which this is legal are discussed in the section Variables in Extent and INITIAL Expressions in Chapter 7.

Multi-dimensional Arrays

The array examples just given were all one-dimensional arrays, meaning that the individual scalar elements of the array were referenced by means of a single subscript. PL/I permits arrays of two or more dimensions as well. For example, consider the following declaration:

```
DECLARE MAT(4,3) FLOAT DECIMAL(7);
```

This statement specifies that MAT is a two-dimensional array (informally called a matrix or table), containing four rows and three columns.

Figure 5-2 illustrates the array MAT. When you use the symbol MAT by itself, you are referring to all 12 scalar elements of the array. By using two subscripts, such as in MAT(4,3), you are referring to a single element of the array.

MAT

MAT (1,1)	MAT (1,2)	MAT (1,3)
MAT (2,1)	MAT (2,2)	MAT (2,3)
MAT (3,1)	MAT (3,2)	MAT (3,3)
MAT (4,1)	MAT (4,2)	MAT (4,3)

The Two-dimensional Array MAT
Figure 5-2

MAT (1, 1)
MAT (1, 2)
MAT (1, 3)
MAT (2, 1)
MAT (2, 2)
MAT (2, 3)
MAT (3, 1)
MAT (3, 2)
MAT (3, 3)
MAT (4, 1)
MAT (4, 2)
MAT (4, 3)

Layout of MAT in Memory
Figure 5-3

For example, the statement

```
PUT LIST(MAT);
```

prints out all 12 elements of the array, in the order MAT(1,1), MAT(1,2), MAT(1,3), MAT(2,1), ..., MAT(4,2), MAT(4,3). (This is known as row-major order. Row-major order is how MAT is arranged in memory, as illustrated by Figure 5-3. FORTRAN programmers should be aware that FORTRAN uses column-major order.) On the other hand, the statement

```
PUT LIST(MAT(3,2));
```

prints only that one scalar element.

The bound specification, as applied to two-dimensional arrays, is used separately with each of the dimensions. The array MAT, which has been described, has a lower bound of 1 for both the first and second dimensions, but has an upper bound of 4 for the first dimension and 3 for the second dimension. As in the case of singly dimensioned arrays, you may use the colon to specify a lower bound other than 1. For example, the statement

```
DECLARE GNP(1950:1970,4) FIXED DECIMAL(12);
```

specifies that the array GNP is a 21x4 two-dimensional array, with lower bounds of 1950 for the first dimension and 1 for the second dimension, and upper bounds of 1970 for the first dimension and 4 for the second dimension.

Although they are very unusual, PL/I permits arrays of more than two dimensions. For example,

```
DECLARE M(8,4,5:3,16) CHARACTER(5);
```

is a declaration for a four-dimensional array, M. The maximum number of dimensions is eight.

Array Cross Sections

Use an asterisk (*) to refer to a particular single row or column of a two-dimensional array. For example, MAT(3,*) refers to the third row of MAT, where MAT is the two-dimensional array described above. The statement

```
PUT LIST(MAT(3,*));
```

prints out the three values, MAT(3,1), MAT(3,2), and MAT(3,3), in the third row of MAT. Similarly, MAT(*,2) refers to the second column of the array MAT.

In multi-dimensional arrays, use an asterisk in any of the subscript positions to specify that all subscripts in that subscript position are to be included. For example, with the four-dimensional array M described above, you may use M(*,I,*,3) to specify a two-dimensional cross section.

Structures

The second basic aggregate type is a structure. Whereas in an array all the scalar elements have the same data type, in a structure the individual scalars may have different data types.

Consider, for example, the following declaration:

```

DECLARE 1 INDIVIDUAL,
        2 NAME CHARACTER(20) VARYING,
        2 AGE FIXED BINARY,
        2 SALARY FIXED DECIMAL(9,2);

```

Note

Commas, not semicolons, are used to separate elements within the structure declaration. Level numbers are explained below.

This declaration specifies that INDIVIDUAL is a structure, and that it contains three individual scalar elements, called members. Each of the structure members has its own identifier. The first member, called NAME, has the attributes CHARACTER(20) VARYING, the second member, called AGE, has the attributes FIXED BINARY, and the third, called SALARY, has the attributes FIXED DECIMAL(9,2). Thus, INDIVIDUAL is a structure containing three scalar element members, with three different data types.

When used by itself, the identifier INDIVIDUAL refers to all three scalar elements in the structure. For example, the statement

```

PUT LIST(INDIVIDUAL);

```

prints out all three of the values.

You may refer to each of the three members separately as INDIVIDUAL.NAME, INDIVIDUAL.AGE, INDIVIDUAL.SALARY. A name specified in this fashion using a period is called a qualified name. You may also use the unqualified identifiers, NAME, AGE, and SALARY, to refer to the individual scalar elements, providing that doing so would not be ambiguous. An example of a situation in which such use would be ambiguous is a program containing declarations for different structures with the same member names. In such a case, use the fully qualified identifiers.

It is possible for a member of a structure itself to be a structure; it is then called a substructure. For example, consider the following declaration:

```

DECLARE 1 SALE
      2 PRODUCT,
        3 SERIAL PIC 'AAX999',
        3 DESCRIP CHAR(20),
      2 DATE,
        3 MONTH CHAR(3),
        3 DAY FIXED DEC(2),
        3 YEAR FIXED DEC(4),
      2 PRICE PIC '$$$$9V.99';
    
```

Here, the major structure name is SALE. Its first member is a substructure called PRODUCT, which, itself, has two members called SERIAL and DESCRIP. The second member of SALE is a substructure called DATE, and the third member is a scalar called PRICE. This structure contains six scalar elements. The first of these can be referenced by means of the qualified name SALE.PRODUCT.SERIAL. You may also use any of the references SERIAL, PRODUCT.SERIAL, or SALE.SERIAL, provided that there is no other declaration in your program that would make such a reference ambiguous.

The numbers 1, 2, and 3 in the declaration above are called level numbers and are used to indicate the depth of the identifier inside the structure or substructure. You need not specify these level numbers consecutively. In fact, the preceding declaration is completely equivalent to the following one:

```

DECLARE 1 SALE,
      3 PRODUCT,
        7 SERIAL PIC 'AAX999',
        7 DESCRIP CHAR(20),
      3 DATE,
        9 MONTH CHAR(3),
        9 DAY FIXED DEC(2),
        9 YEAR FIXED DEC(4),
      3 PRICE PIC '$$$$9V.99';
    
```

The only requirement is that the level number of a member must be greater than the level number for its structure name.

The maximum number of items in a structure is 1024.

The BY NAME Option

If two structures have one or more member elements of the same name, the assignment statement using the BY NAME clause moves the values of like-named sources to targets. Consider the following two structures and the assignment statement

```

DECLARE 1 A,
        2 B,
        2 C,
        3 Y,
        2 D;
DECLARE 1 X,
        2 B,
        2 Y,
        2 D;
A = X, BY NAME;

```

This statement moves values between elements whose names are found in the same level of both structures. The result is that B and D of structure A have the values of the like-named variables of X. The value of Y in structure A is not changed.

Arrays of Structures

An array of structures is an array, each of whose elements is a structure. The structure INDIVIDUAL, which was described above, can be made into an array of structures as follows:

```

DECLARE 1 INDIVIDUAL(50),
        2 NAME CHAR(20) VAR,
        2 AGE BIN FIXED,
        2 SALARY FIXED DEC(9,2);

```

INDIVIDUAL is here an array of 50 elements, each of which is a structure containing three numbers. Therefore, INDIVIDUAL contains 150 scalar elements. You can reference various cross sections and individual elements of this array of structures as follows:

- INDIVIDUAL(5) is the fifth structure in the array. It contains three scalar elements.
- INDIVIDUAL(5).NAME, INDIVIDUAL(5).AGE, and INDIVIDUAL(5).SALARY are the three scalar elements of INDIVIDUAL(5). In referencing a single scalar element in an array of structures, you may move the subscript to the right of the qualified name. Thus, for example, PL/I considers INDIVIDUAL.SALARY(8) to be equivalent to INDIVIDUAL(8).SALARY.

- `INDIVIDUAL.NAME` is a cross section array containing 50 elements, which are, the `NAME` fields in each of the 50 structures in the array of structures.

THE ALIGNED AND UNALIGNED ATTRIBUTES

Use the `ALIGNED` and `UNALIGNED` attributes to specify whether the storage area occupied by a variable is to be aligned on a word boundary or not.

The purpose of these attributes is to permit you to specify what kind of optimization criteria you wish to follow. If you specify that data is to be `ALIGNED`, it may occupy more storage area, but accessing it is faster; on the other hand, `UNALIGNED` data can be packed together to save space, but is harder to access.

BIT ALIGNED and UNALIGNED Data

Normally, PL/I stores `BIT NONVARYING` data in whatever storage space is available, whether aligned on a word boundary or not. This means that several `BIT NONVARYING` data areas might be packed together, making the data hard to access. If you specify `ALIGNED` with the `BIT` attribute, the PL/I aligns the `BIT` string on a word boundary, so that the data can be accessed faster.

Consider the following declarations:

```
DECLARE BARRAY(200) BIT(2) ALIGNED;  
DECLARE BARRAY2(200) BIT(2) UNALIGNED;
```

Both `BARRAY` and `BARRAY2` are `BIT` arrays, each containing 200 data elements, where each of the data elements is a string containing 2 bits. However, each of the 200 data elements of `BARRAY` is aligned on a word boundary, so that `BARRAY` occupies 200 words of storage. On the other hand, `BARRAY2` is `UNALIGNED`, and so the bits of the data elements are packed together. This means that `BARRAY2` occupies a total of 400 bits, or 25 16-bit words.

If you do not specify either `ALIGNED` or `UNALIGNED` for a `BIT NONVARYING` declaration, PL/I uses a default of `UNALIGNED`.

ALIGNED and UNALIGNED With Other Data Types

PL/I gives every data variable either the ALIGNED attribute or the UNALIGNED attribute. The default is UNALIGNED for BIT NONVARYING, CHARACTER NONVARYING, and PICTURE data; otherwise the default is ALIGNED. You may override these defaults by specifying either ALIGNED or UNALIGNED in the declaration.

THE DEFINED ATTRIBUTE

The DEFINED attribute allows you to specify that the storage allocated for one variable is to be shared by another variable. One use of this attribute is to save storage by using a single area for two separate large aggregates.

However, that is not its main purpose. The DEFINED attribute is most useful in providing you a convenient means of referencing the same storage area in two different ways. For example,

- You may reference the same storage area by two different names, or you may represent a portion of one variable's storage area by a different name.
- You may reference the same storage area either as one long CHARACTER string or as an aggregate containing several shorter CHARACTER strings.
- You may specify that the elements of an array are to be automatically referenced in a different order from the one in which they are stored in memory.

The following pages cover the different uses of the DEFINED attribute: simple defining, string overlay defining, and ISUB defining.

Simple Defining

Consider the following declarations:

```
DECLARE X FIXED;
DECLARE Y FIXED DEFINED(X);
```

These statements specify that X and Y have the same data type, FIXED, and that Y is to occupy the same storage area as X. The result is that any reference to Y in any statement is the same as a reference to X in that statement.

A more useful example is illustrated by the following declarations:

```
DECLARE A(100);  
DECLARE B DEFINED(A(23));
```

In this example, the variable B is specified as occupying the same storage as the twenty-third element of the array A. An even more sophisticated example is the following:

```
DECLARE C DEFINED(A(K));
```

Like B, C is DEFINED on a single element of the array A. In this case however, the number of the element is determined by the value of K. K is reevaluated each time C is referenced, and so C might be equivalent to any of the elements of A.

You may also use simple defining to specify that one variable is to be used as a singly dimensioned cross section of a two-dimensional array. Consider, for example, the following declarations:

```
DECLARE MAT(10, 20);  
DECLARE ROW3(10) DEFINED(MAT(3,*));  
DECLARE ROW(10) DEFINED(MAT(K,*));
```

In this example, ROW3 is a singly dimensioned array equivalent to the third row of the doubly dimensioned array MAT. ROW is another singly dimensioned array, equivalent to the Kth row of MAT, where the value of K is reevaluated each time ROW is referenced.

CHARACTER String Overlay Defining

Using string overlay defining, you can reference the storage area occupied by an UNALIGNED CHARACTER NONVARYING string by means of an aggregate whose elements are all UNALIGNED CHARACTER NONVARYING strings. Alternatively, you can reference the storage occupied by one aggregate of UNALIGNED CHARACTER NONVARYING strings by means of another such aggregate.

Consider the following declarations:

```
DECLARE C CHARACTER(100);  
DECLARE D(100) CHARACTER(1) DEFINED(C);
```

DATA TYPES AND DATA ATTRIBUTES

The value of C is stored as 100 individual characters packed together into a storage area. The declaration of D specifies that its 100 characters are to be the same as the 100 characters of C. Therefore, for example, a reference to SUBSTR(C,5,1) is the same as a reference to D(5).

A DEFINED string need not overlay the entire storage area of the variable over which it is DEFINED. For example,

```
DECLARE D2(20) CHARACTER(2) DEFINED(C);
```

specifies that D2 is to use as its storage area the first 40 characters of the 100-character storage area occupied by C. It is not necessary for the overlaid storage area to be at the beginning of the storage area being overlaid. For example,

```
DECLARE D3(20) CHARACTER(2) DEFINED(C) POSITION(13);
```

specifies that D3 is to occupy the same storage as C, starting at the thirteenth character of C. Since D3 occupies 40 characters of storage, it uses characters 13 through 52 of the storage area occupied by C.

A useful application of string overlay defining is to break up a long character string into separate fields, each of which is meaningful in its own right. For example, consider the following declarations:

```
DECLARE CARD CHARACTER(80);  
DECLARE 1 S DEFINED(CARD),  
        2 NAME CHARACTER(30),  
        2 ADDRESS CHARACTER(20),  
        2 CITY CHARACTER(30);
```

In this example, the variable CARD contains a card image of 80 characters. The first 30 columns of the card contain a person's name, the next 20 characters contain the address, and the last 30 characters contain the city. By using string overlay defining, you can reference the individual portions of the CHARACTER string CARD without having to perform unnecessary copying operations from one variable to another.

As a final example, it is possible for one aggregate to be overlaid by another. Consider the following declarations:

```
DECLARE 1 S(5),
        2 T CHARACTER(3),
        2 U(6),
          3 V CHARACTER(1),
          3 W CHARACTER(20);
DECLARE SD(129) CHARACTER(5) DEFINED(S);
```

S is an array of structures whose individual data elements are all UNALIGNED CHARACTER NONVARYING strings. The total storage area for S contains 645 characters. SD is an array aggregate, also containing 645 characters, which shares the same storage space as S.

For the purpose of string overlay defining, the PICTURE data type may be overlaid in the same way that an UNALIGNED CHARACTER NONVARYING string may be overlaid. Consider the following declarations:

```
DECLARE SALARY PICTURE '$$, $$9V.99';
DECLARE SAL_STRING CHARACTER(9) DEFINED(SALARY);
```

Both SALARY and SAL_STRING occupy the same nine characters of storage.

BIT String Overlay Defining

UNALIGNED BIT NONVARYING strings may be defined as string overlays in the same way that UNALIGNED CHARACTER NONVARYING strings may be overlaid. All the examples of the preceding subscripts are valid with CHARACTER replaced by BIT, except that of course a BIT string may not meaningfully overlay a PICTURE variable.

iSUB Defining

Sometimes you wish to reference the elements of an array in a different order from the one in which they are stored in memory. You may do this conveniently by means of iSUB defining. The expression iSUB refers to one subscript or dimension of the area to be overlaid, and i is the number of the subscript. Consider the following declarations:

```
DECLARE A(100);
DECLARE B(100) DEFINED(A(101 - 1SUB));
```

The declaration of B says that a reference to an element of the array B is equivalent to a reference to some element of the array A. In the declaration, 1SUB stands for the subscript used in the reference to B. For example, a reference to B(K + L) is equivalent to a reference to A(101 - (K + L)). Therefore, for example, a reference to B(1) is equivalent to a reference to A(100), and a reference to B(100) is equivalent to a reference to A(1). In fact, the array B is simply the array A in reverse order.

A slightly more complicated example is the following:

```
DECLARE M(40,40);
DECLARE N(40,40) DEFINED(M(2SUB, 1SUB));
```

In this example, M and N are two-dimensional arrays occupying the same storage. A reference to an element of N is replaced by a reference to M with the subscripts reversed, because, in the declaration, the element 2SUB stands for the second subscript expression in the reference to N, and 1SUB stands for the first subscript reference.

You may define a singly-dimensioned array over the diagonal of the two-dimensional array M as follows:

```
DECLARE DIAG(40) DEFINED(M(1SUB, 1SUB));
```

For example, a reference to DIAG(K) is equivalent to a reference to M(K,K).

THE LIKE ATTRIBUTE

When you wish to specify that one structure or substructure has the same members as another structure or substructure, you may simplify your declaration by using the LIKE attribute. Consider the following declarations:

```
DECLARE 1 A,
        2 X FIXED,
        2 Y FLOAT;
DECLARE 1 B LIKE A;
```

The declaration of B specifies that it is to be a structure with the same members as A. These members have the same names, aggregate types, and data types. Therefore, the above declaration of B is equivalent to the following:

```
DECLARE 1 B,  
        2 X FIXED,  
        2 Y FLOAT;
```

When you specify that one structure is LIKE another structure, you are specifying only that they have the same members; the attributes of the structure (such as dimensioning or the storage class) need not be the same. For example,

```
DECLARE 1 PRODUCTS(1000) CONTROLLED,  
        2 NAME CHARACTER(20) VARYING,  
        2 STOCK_NUM PICTURE 'A999',  
        2 PRICE FIXED DECIMAL(7,2);  
DECLARE 1 ITEM STATIC LIKE PRODUCTS;
```

specifies that ITEM is to be a structure with the same members as PRODUCTS. Note, however, that PRODUCTS is an array of structures while ITEM is a single structure, and that PRODUCTS has the CONTROLLED storage class attribute, while ITEM has the STATIC storage class attribute. (Storage class attributes are defined and discussed in Chapter 7.)

THE INITIAL ATTRIBUTE

Abbreviation: INIT for INITIAL

You must assign a value to each PL/I variable before you use that variable in any other way. A convenient way to give an initial value to a variable is to use the INITIAL attribute when you declare the variable, to specify an initial value to be assigned to it.

Initializing Scalars

The following examples illustrate how to use the INITIAL attribute to provide initial values for scalar variables:

```
DECLARE X FLOAT INIT(0);  
DECLARE Y FIXED DECIMAL(5,2) INIT(7.3);  
DECLARE C CHARACTER(200) VARYING INIT('');
```

The variable X is initialized to 0, the variable Y is initialized to 7.3, and the string variable C is initialized to the null string. Like other attributes, the INITIAL attribute may be factored in a DECLARE statement. For example,

```
DECLARE(U,V) INITIAL(0);
```

can be used to initialize both U and V to 0.

Initializing Arrays

In order to initialize an array, specify an initial value for each of the elements of the array. For example,

```
DECLARE A(5) INITIAL(8, 7, 4, 25, -15);
```

specifies that A is to be an array of five elements, and the initial values of the array elements are 8, 7, 4, 25 and -15, respectively.

If you wish to initialize all the elements of the array to the same value, you may use a repetition factor, as in the following example:

```
DECLARE B(10) FLOAT INITIAL((10)0);
```

Here, B is an array of 10 elements, all of which are initialized to 0.

For more complicated initializations, use repetition factors as often as they are needed. Consider, for example, the following:

```
DECLARE C(100) INITIAL((25)0, 1, 2, 3, 4, 5, (70)-1);
```

In this example, C is an array of 100 elements. The first 25 of these elements are initialized to 0, elements 26 through 30 are initialized to 1, 2, 3, 4 and 5, respectively, and elements 31 through 100 are initialized to -1.

You do not have to initialize the entire array. For example, the declaration

```
DECLARE D(10) INITIAL((5)0);
```

initializes the first five elements of D to 0 and leaves the last five elements uninitialized.

Use an asterisk to specify explicitly that an element of the array is to be left uninitialized. For example,

```
DECLARE E(10) INITIAL(*,(4)0,(4)*,1);
```

specifies that the first element of E is to be left uninitialized, the next four elements are initialized to 0, the next four elements are uninitialized, and the last element is initialized to 1.

A special consideration must be followed when you initialize arrays of CHARACTER or BIT strings. Suppose you wish to declare a CHARACTER string array called STR, and you wish to initialize each of the elements of the array to the string 'A'. You might mistakenly try the following declaration:

```
DECLARE STR(10) CHARACTER(100) VARYING INITIAL((10)'A');
```

However, PL/I misinterprets this statement as written. PL/I interprets (10)'A' as the character string constant equal to 'AAAAAAAAAA'. PL/I initializes STR(1) to this string, and leaves the other elements of STR uninitialized. The proper declaration is the following:

```
DECLARE STR(10) CHARACTER(100) VARYING INITIAL((10)('A'));
```

The extra set of parentheses around 'A' permits PL/I to interpret the statement as you intend.

Initializing Structures and Arrays of Structures

To initialize a structure, specify the INITIAL attribute for each element of the structure. For example, the declaration

```
DECLARE 1 S,  
      2 A FIXED INITIAL(0),  
      2 B FLOAT INITIAL(5);
```

specifies that the structure element S.A is to be initialized to 0 and S.B is to be initialized to 5.

To initialize an array of structures, remember that each of the members of the structure inherits the dimensions of the structure. Therefore, use the conventions described above under Initializing Arrays to specify an initial value for each of the elements of the member arrays. For example,

```
DECLARE 1 T(100),
        2 A FIXED INITIAL((100)0),
        2 B FLOAT INITIAL((50)5);
```

specifies that all 100 elements of the array T.A are to be initialized to 0, and the first 50 elements of the array T.B are to be initialized to 5. The last 50 elements of T.B are left uninitialized.

Using Variables in the INITIAL Attribute

A statement like

```
DECLARE COUNT(1000) INITIAL((K)(M+3));
```

is legal under certain circumstances, as described in Chapter 7. If the rules in that chapter are followed, then when COUNT is allocated, the values of K and M are determined, and the first K elements of the array COUNT are initialized to the value of M + 3.

THE DEFAULT STATEMENT

Abbreviation: DFT for DEFAULT

This is a rarely used statement. Use it to specify default attributes and default rules to override the automatic PL/I defaults.

Let's start with some examples:

- DEFAULT(RANGE(A:H) ! RANGE(0:Z))FLOAT;

Normally, if you use an undeclared variable in your program, or if you declare a numeric variable but don't specify either FIXED or FLOAT, then PL/I gives it the default attribute FIXED. This was not true in older implementations of PL/I, however. Older implementations followed the "I through N rule," which gave variables beginning with the letters I, J, K, L, M, or N the

default attribute FIXED and all other variables the default attribute FLOAT.

The above DEFAULT statement specifies that you wish to follow the I through N rule. In this statement, RANGE(A:H) refers to any variable beginning with the letters A through H. The phrase RANGE (O:Z) refers to all variables beginning with the letters O through Z. The exclamation point is the symbol for OR. Therefore, this DEFAULT statement specifies that all variables beginning with the letters A through H or O through Z should have the default attribute FLOAT. This means that all variables beginning with I through N are still given the system default attribute, FIXED.

The above example also makes the variable DEFAULT compatible with IBM PL/I.

- DEFAULT (VARIABLE) STATIC;

In this statement, the keyword VARIABLE refers to any variable in your program. This DEFAULT statement says that every variable of your program should have the default attribute STATIC, unless declared otherwise. (STATIC is a storage class attribute; the default storage class attribute is AUTOMATIC. Storage attributes are described in Chapter 7.)

- DEFAULT (FIXED & DECIMAL) PRECISION(7, 2);

This statement specifies that any variable with both the attributes FIXED and DECIMAL is to be given a default precision of (7, 2), instead of the system default precision (5, 0).

If your program contains a DEFAULT statement, it is important to remember that the statement only specifies default attributes. You may override these defaults by declaring any variable explicitly with the attributes that you want it to have. For example, if you declared the variable COUNT with the FIXED attribute, the default attribute FLOAT would not apply. FLOAT applies only when you do not specify either FIXED or FLOAT.

A second important point to remember is that the default attributes apply only when they are consistent with the other attributes of the variable. For example, if a program containing the first DEFAULT statement above contained the declaration

```
DECLARE STRING CHARACTER(5);
```

then PL/I would not give the default attribute FLOAT to STRING, since FLOAT is inconsistent with the explicitly declared attribute CHARACTER.

Format of the DEFAULT Statement

The DEFAULT statement has the following format:

```
DEFAULT (attribute-test) attribute-list [, attribute-list...];
```

This statement specifies that PL/I is to apply the attribute-test to each of the variables in your program and use the default attribute in each of the attribute-lists, where consistent, for each of the variables meeting the attribute-test.

The attribute-test is a logical combination of keywords testing either attributes that the variable already has or the letters in the name of the variable. The keywords that test attributes are the following:

ALIGNED	DEFINED	INTERNAL	PRECISION
AREA	DIMENSION	KEYED	PRINT
AUTOMATIC	DIRECT	LABEL	REAL
BASED	ENTRY	LOCAL	RECORD
BINARY	ENVIRONMENT	MEMBER	RETURNS
BIT	EXTERNAL	NONVARYING	SEQUENTIAL
BUILTIN	FILE	OFFSET	STATIC
CHARACTER	FIXED	OPTIONS	STREAM
COMPLEX	FLOAT	OUTPUT	STRUCTURE
CONDITION	FORMAT	PARAMETER	UNALIGNED
CONSTANT	GENERIC	PICTURE	UPDATE
CONTROLLED	INITIAL	POINTER	VARIABLE
DECIMAL	INPUT	POSITION	VARYING

In addition to the attribute-test keywords, you may use the keyword RANGE to test the letters in the variable name. The RANGE keyword has two forms. The first is

```
RANGE(letter:letter)
```

It tests whether the first character in the variable name lies within the specified range of letters. The form

```
RANGE(letters)
```

tests whether the variable identifier name begins with the specified string of one or more letters.

You may combine these keyword tests into a full attribute-test by using the logical operators & for AND, ! or | for OR and ^ for NOT. For example, the statement

```
DEFAULT (RANGE(B:C) & (BIT ! CHAR) & ^ RANGE(CNV)) VARYING;
```

specifies that any variable beginning with the letters B or C and having either the BIT or CHARACTER attribute, but not beginning with the letters CNV, should be given the default attribute VARYING, unless declared otherwise.

The attribute keywords above test whether the variable has any attribute using that keyword. For example, including the keyword PRECISION tests whether any precision attribute has been specified for the variable.

Use the DEFAULT statement to change the default length of strings. For example,

```
DEFAULT CHAR CHAR(17);
```

changes the default length of character strings to 17.

Your program may contain as many DEFAULT statements as you wish. PL/I applies the default statements in the order in which they appear in your program. After PL/I has applied all the DEFAULT statements in your program, it applies the system defaults.

It is legal for a DEFAULT statement to appear inside an internal PROCEDURE or BEGIN block. In that case, the default rules apply only to explicit declarations made within that block.

Compiler Application of the DEFAULT Statement

When you DECLARE a variable, you usually specify only a few of the attributes and let PL/I apply the remaining ones.

For each variable in your program, PL/I forms a set of attributes for that variable. PL/I starts with the attribute you specify in the DECLARE statement, and uses the DEFAULT statements to add attributes as follows:

1. PL/I uses the attribute-test in the DEFAULT statement to test whether the attribute set so far satisfies the test. If so, PL/I continues with the next step.

2. For each of the attribute-lists in the DEFAULT statement, if that list of attributes is consistent with the attribute set so far, then PL/I adds the new attribute-list to the attribute set.

PL/I performs these steps for each of the DEFAULT statements in your program.

Other Forms of the DEFAULT Statement

The form

```
DEFAULT SYSTEM;
```

specifies that the standard system default rules are to apply. Use this statement in an inner block if you do not wish that block to inherit the effects of a DEFAULT statement specified in an outer block.

Another form of the DEFAULT statement is

```
DEFAULT (attribute-test) ERROR;
```

to specify that certain attributes or combinations of attributes are to be illegal in your program.

The form

```
DEFAULT NONE;
```

specifies that no defaults are applied, not even system defaults.

"P"-Constants and the DEFAULT Statement

Default attributes apply to all objects within their scope, including constants. Consider the following code segment:

```
DEFAULT (DECIMAL) FLOAT;
DECLARE X DECIMAL(2);
X = 10;
```

Under the system default attributes, the constant 10, which has by its nature the attribute CONSTANT REAL DECIMAL(2), acquires the scale attribute FIXED. However, in this example, the DEFAULT statement establishes that any object with the attribute DECIMAL has FLOAT scale. Thus, 10 as well as X has the scale attribute FLOAT.

This feature is convenient in that it prevents unnecessary routine conversions. Sometimes, though, it is equally convenient to force a constant to be interpreted according to system default attributes. This can be done by appending a P after the constant, as in the following example:

```
DEFAULT (DECIMAL) FLOAT;
DECLARE X DECIMAL(2);
DECLARE Y FIXED DECIMAL(5);
X = 10; /* 10 is FLOAT DECIMAL(2) */
Y = 00010P; /* 00010P is FIXED DECIMAL(5) */
```

6

Evaluating Expressions

EXPRESSIONS, DATA CONVERSIONS, AND AGGREGATE PROMOTIONS

This chapter describes how PL/I evaluates expressions. It first defines a PL/I expression, then discusses the following particular aspects of the evaluation process:

- Data types. If you use a PL/I expression of one data type in a context that requires a different data type, PL/I must convert the value of your expression to the data type required by the context. In addition, it may be necessary for PL/I to perform conversion during evaluation of the expression to produce intermediate results. This would happen, for example, if you attempted to multiply a CHARACTER string value by a BIT string value. This chapter covers the detailed rules for such conversions.
- Aggregates. PL/I permits an expression to contain variables that are aggregates (arrays or structures), with the result that the entire expression has an aggregate value. An aggregate expression has an aggregate type of scalar, array, structure, or some combination of these. If an expression of one aggregate type is used in a context that requires a different aggregate type, an aggregate promotion must take place. This chapter discusses the rules for aggregate promotions.

Almost all computer programs perform computations of some sort on data values. In PL/I, the computation of new data values from old is done

by means of expressions. Expressions appear in almost all PL/I statements. Consider, for example, the following assignment statement:

```
PARAMETER = 2 * (LENGTH + WIDTH);
```

This statement contains a variable, PARAMETER, to the left of the equal sign, and an expression to the right of the equal sign. The statement specifies that the value of the expression is to be computed and assigned to the variable on the left. Another example is

```
PUT LIST(X + Y);
```

Here, the expression X + Y is to be computed, and its value is to be printed. As a final example, consider the following:

```
IF VALUE > 5 THEN GO TO BIG;
```

This IF statement contains an example of what is called informally a logical expression. The expression VALUE > 5 is to be evaluated to obtain what is informally called a truth value, in order to determine whether control should transfer to the statement with label BIG. More precisely, VALUE > 5 is a PL/I expression whose value will have the data type BIT(1), which is the PL/I equivalent of what are called logical data types in other languages.

FORMING EXPRESSIONS

An expression is composed of the following basic elements:

1. Variables and constants
2. Operators, such as +, *, or &
3. Parentheses
4. Built-in functions

The methods for referencing variables and constants are discussed in Chapter 5. This section describes the other components of a PL/I expression.

Arithmetic Operators

Arithmetic operators are those whose operands must be arithmetic (numeric), and whose results are also arithmetic. There are seven arithmetic operators, and they may be grouped as follows:

1. The four infix operators are the plus sign (+), minus sign (-), asterisk (*), and slash (/). These operators stand for addition, subtraction, multiplication, and division, respectively, of two operands.
2. The prefix operators are + and -. Infix minus is usually called subtraction, while prefix minus is usually called negation. The expression `-(A - B)` illustrates the difference. The first minus sign in this expression is prefix minus, because it has only one operand, `(A - B)`, and it operates by simply reversing the sign of that operand. The second minus sign is infix minus, because it has two operands, `A` and `B`, and it operates by subtracting the second operand from the first.
3. The exponentiation operator, `**`. This operator takes two operands, and operates by raising the first operand to the power of the second. For example, the expression `X**3` is usually called `X` cubed, and has the value `X * X * X`.

The Comparison Operators

There are eight comparison operators, as shown in Table 6-1.

Table 6-1
Comparison Operators

Operator	Meaning
<code>=</code>	Equals
<code>^=</code>	Does not equal
<code><</code>	Is less than
<code><=</code>	Is less than or equal
<code>></code>	Is greater than
<code>>=</code>	Is greater than or equal
<code>^<</code>	Is not less than
<code>^></code>	Is not greater than

PL/I Reference Guide

Comparison operators are used most frequently in the IF statement, in order to make some decision based on a comparison of two values. For example, the statement

```
IF A = B THEN STOP;
```

contains the comparison $A = B$. The statement specifies that if A equals B, the program should stop.

However, you should realize that the comparison operators can be used in expressions in any context. A comparison operator is an operator that takes two operands, and that operates by comparing the two operands and producing a BIT(1) result. This result has the value '1'B if the comparison was true, and the value '0'B if the comparison was false. Thus, for example, a program may contain the following statements:

```
DECLARE TEST BIT(1);  
..  
TEST = A = B;  
..  
IF TEST THEN STOP;
```

The middle statement in this example is an assignment statement with the variable TEST on the lefthand side, and the expression $A = B$ on the righthand side. PL/I executes this assignment statement by comparing the variables A and B for equality, and setting TEST to either '0'B or '1'B, depending upon the result of the comparison. The last statement of this example is an IF statement that uses the variable TEST as an operand.

Note

In Prime ED, the operator \wedge must be entered in duplicate, as $\wedge\wedge$.

Logical Operators

The logical operators, ampersand (&), vertical bar (|), exclamation point (!), and caret (^), are shown in Table 6-2.

Table 6-2
Logical Operators

Operator	# of Operands	Meaning
&	2	"And"
	2	"Or"
!	2	"Or"
^	1	"Not"

Like the comparison operators, the logical operators are used most frequently in the IF statement. In fact, usually the logical operators are used with the comparison operators in order to provide for the testing of two or more comparisons. For example, the statement

```
IF (CASE > 2 & CASE <= 5) | CASE = 23 THEN CALL SMALL;
```

contains three comparisons, joined by means of the operators & and |. The statement says that if CASE equals 23, or is greater than 2 and less than or equal to 5, then control should pass to the procedure SMALL. An example using the ^ operator is

```
IF ^ (X > 0 & X < .0001) THEN GO TO LARGE;
```

which states that if it is not true that X is both positive and less than .0001 (which is the same as saying that either X is negative or zero or greater than or equal to .0001), then control should pass to the statement LARGE.

Precisely speaking, the & and | each take two BIT operands and produce a BIT result. The ^ operator takes a single BIT operand and computes a BIT result. Like the comparison operators, the logical operators can be used in any expression that can accommodate a BIT result.

Note

In Prime ED, the operator NOT must be entered as ^^.

Concatenation

Use the concatenation operator || or !! with two strings, either CHARACTER or BIT, in order to stick them together (the precise term is concatenate) to form one long string.

PL/I Reference Guide

Examples of PL/I concatenation for CHARACTER strings and BIT strings follow:

```
DCL A CHAR(4);          DCL D BIT(2);
DCL B CHAR(5);          DCL E BIT(1);
DCL C CHAR(10) VARYING; DCL F BIT(3);
.
.
A = 'SOME';            D = '01'B;
B = 'THING';           E = '1'B;
C = A || B;            F = D !! E;
```

C has been assigned the CHARACTER string 'SOMETHING', and F has been assigned the BIT string '011'B.

Operator Priority and Parentheses

Consider the following three assignment statements:

```
A = (2 * B) + C;
A = 2 * (B + C);
A = 2 * B + C;
```

These three assignment statements are identical except for their use of parentheses. These parentheses specify the order in which the operations are to be performed. In the first assignment statement, multiplication is to be performed first. In the second assignment statement, addition is to be performed first.

In the absence of parentheses, PL/I performs multiplication before addition, so that the third assignment statement is equivalent to the first. This is precisely what one would expect from the usual rules of algebra.

Adjacent operators are two operators in an expression separated by a single operand, with all parentheses between the two operands completely matched. For example, in the expression

```
2 * B + C;
```

the * and the + are adjacent operators separated by the operand B. In the expression

```
A * (B + C) - D;
```

EVALUATING EXPRESSIONS

the * and the - are adjacent operators separated by the single operand (B + C). The + operator is not considered adjacent to either of the other operators, since, in each case, there is an unbalanced parenthesis separating them.

PL/I provides precise rules for determining, given two adjacent operators, which operation will be performed first. Table 6-3 summarizes these rules. This table breaks up the collection of PL/I operators into seven different priority levels, with 1 the highest priority. The rules for the order of evaluation of two adjacent operators in an expression are as follows:

1. If the two operations are at different priority levels in Table 6-3, the operation with higher priority is performed first.
2. If the two operations are at the same priority level in the table, they are performed in the order indicated by the rightmost column of the table. That is, at the top priority level, the one on the right is performed first, while at the other priority levels, the one on the left is performed first.

Table 6-3
Operator Priority

Level	Operators	Ordering between operators at this level
1	Prefix +, prefix -, **, ^	Right-to-left
2	*, /	Left-to-right
3	Infix +, Infix -	Left-to-right
4	, !!	Left-to-right
5	Comparison operators =, ^=, >, <, >=, <=, ^>, ^<	Left-to-right
6	&	Left-to-right
7	!, !	Left-to-right

For example, in the expression

2 * B + C;

since * is at level 2 while + is at level 3, the * is performed first.

In the expression

$$A + B - C;$$

the two operators, + and -, are both at level 3 in the table. As indicated by the table, these two operators will be performed from left to right, so that the + will be performed before the - operator. In the expression

$$-A**B;$$

the - and ** operators are both at level 1, and, as indicated by the table, are evaluated in right to left order. For this reason, the ** operator is performed before the - operator.

An important consequence of this rule is that some operator priorities are not specified. For example, in the expression

$$A * B + C/D$$

it is easy to see that the + operator is performed after either the * operator or the / operator. However, there is no way of determining, by PL/I rules, whether the * operator is performed before or after the / operator. The reason is that * and / are not adjacent operators in this expression. In actual practice, the PL/I compiler uses various optimization techniques to determine, statement by statement, the order in which nonadjacent operators are computed.

Built-in Functions

Another important component of expressions is the built-in function. For example, in the statement

$$X = 3 + \text{SQRT}(B + C);$$

the function SQRT, which computes the square root of its argument, is used. A complete list of built-in functions is provided in Chapter 14.

Notice that built-in functions can be used in any expression, and the argument of a built-in function may itself be an expression.

SCALAR TARGETS AND DATA CONVERSIONS

When an expression having one data type is used in a context that requires a different data type, PL/I must convert the value of your expression to the data type required by the context. This section explains precisely when these conversions are required and what the rules for the conversion are. In addition, precise definitions of the data types of expressions are given.

Examples of Need of Conversions

Suppose the following statements appear in your program:

```

DECLARE A FLOAT, I FIXED;
..
A = I + 3;

```

PL/I computes the expression on the righthand side of the assignment statement by adding 3 to the value of I to get a FIXED result. Since this value is to be assigned to a FLOAT variable, PL/I must convert the FIXED value to FLOAT, before the assignment can be made. In general, PL/I computes the value of the expression on the right-hand side of the assignment statement, and converts it to the data type of the variable on the left-hand side of the assignment statement. This is an example of how the context of an expression can require a conversion.

Another example occurs with the IF statement. The expression following the IF keyword must be a BIT string value. If it is not, PL/I converts it to BIT. For example, if your program contains the statement

```

IF I + 3 THEN GO TO L;

```

then PL/I evaluates the statement I + 3, and then converts it to a BIT string value. Notice that this is an example of a conversion that is not recommended. The rules for numeric to string conversion are quite precise, and are described later in this chapter, but they are very complicated and are full of traps for the unwary.

Arguments to built-in functions often have certain data type restrictions. For example, if you use a FIXED argument with the SQRT built-in function, PL/I must convert the argument to FLOAT. A final example: if an array subscript expression is not fixed, PL/I converts it to FIXED.

Use of Intermediate Targets

The mechanism PL/I uses to evaluate an expression involves the use of intermediate targets. In the evaluation of an expression, PL/I stores the result of each intermediate computation in a temporary location called an intermediate target, with a data type determined by specific rules.

The following program segment illustrates this concept:

```
DECLARE (A,B,C) FIXED DECIMAL(5);  
..  
A = 3;  
B = 12;  
C = 20 * A + B;
```

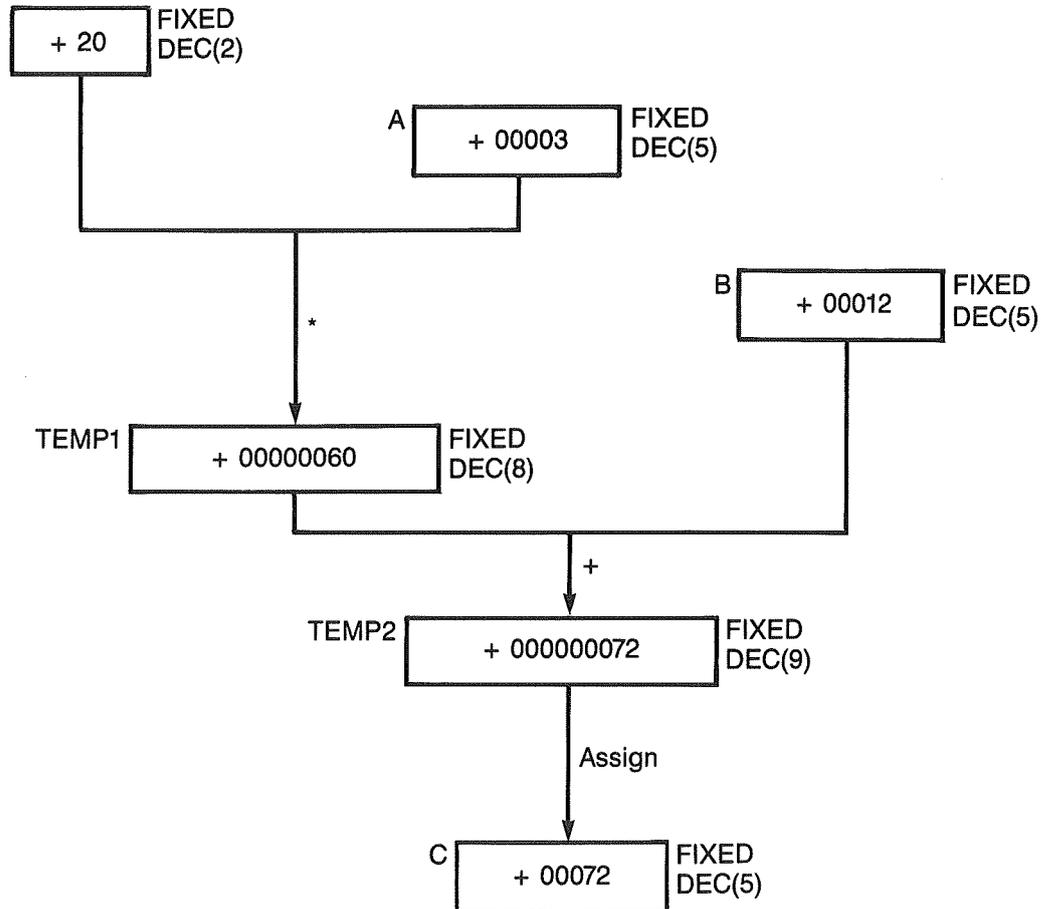
The method PL/I uses to evaluate the last assignment statement is illustrated in Figure 6-1. The boxes in this figure represent storage locations containing the values of the constants, variables, and intermediate values in the computation of this assignment statement.

As this figure illustrates, the constant 20 is multiplied by the value of A, and the result is stored in the intermediate target, which for convenience we name TEMP1. The contents of that storage location are then added to B, and the result is stored in TEMP2. This value is then assigned to C.

As this figure also shows, each of these storage locations has a data type, which is printed to the right of each box. The data type of the constant 20 is FIXED DECIMAL(2), and the data type printed next to the boxes for each of the variables A, B, and C is FIXED DECIMAL(5), as declared. TEMP1 has a data type of FIXED DECIMAL(8), and TEMP2 has a data type of FIXED DECIMAL(9), for this reason: the data type of an intermediate target of an operation depends only upon the operation and the data types of the operands. It does not depend upon the value of the operands.

The data type of TEMP2 is somewhat easier to understand than that of TEMP1. TEMP2 is an intermediate target resulting from the addition of two values, one of which is FIXED DECIMAL(8) and the other FIXED DECIMAL(5). PL/I reasons as follows: TEMP2 must be large enough to accommodate any possible value obtained by adding the two operands. Both operands are FIXED DECIMAL, so TEMP2 is FIXED DECIMAL. To determine the precision, PL/I sees that the first operand has a data type of FIXED DECIMAL(8). Therefore, the maximum value that the first operand can have is +99999999. The second operand is FIXED DECIMAL(5), and so its maximum value is +99999. Therefore, the maximum value that TEMP2 can be expected to accommodate is $99999999 + 99999 = 100099998$.

Since this maximum possible value has nine digits, we make the precision of TEMP2 9. Thus, TEMP2 is FIXED DECIMAL(9). It has a scale factor of zero because both operands have a scale factor of 0.



Intermediate Targets
Figure 6-1

PL/I derives the data type of TEMP1 as follows: TEMP1 is the intermediate target of a multiplication operation, where the two operands are FIXED DECIMAL(2) and FIXED DECIMAL(5). The data type of TEMP1 is FIXED DECIMAL. To determine the precision, PL/I applies the same reasoning as for addition. The first operand can have a maximum value of +99, and the second operand can have a maximum value of +99999. (Of course, the first operand is the constant 20 and cannot have a maximum value of anything other than 20. But don't forget the general rule: the data type of the intermediate target depends only upon the data types of the operands, and not on their values.) Therefore, the maximum value that TEMP1 will have to accommodate is $99 * 99999$ or 9899901, which has seven digits. Therefore, it would seem that the precision of TEMP1 should be seven. However, 1 is added, for the following somewhat obscure but nonetheless very important reason:

the rule for assigning the precision of an intermediate target must take into account the possibility that the operands will be COMPLEX rather than REAL. It is possible to multiply a FIXED DECIMAL(2) COMPLEX operand by a FIXED DECIMAL(5) COMPLEX operand, and get a result requiring eight digits in the real part or the imaginary part. Therefore, TEMP1 is FIXED DECIMAL(8).

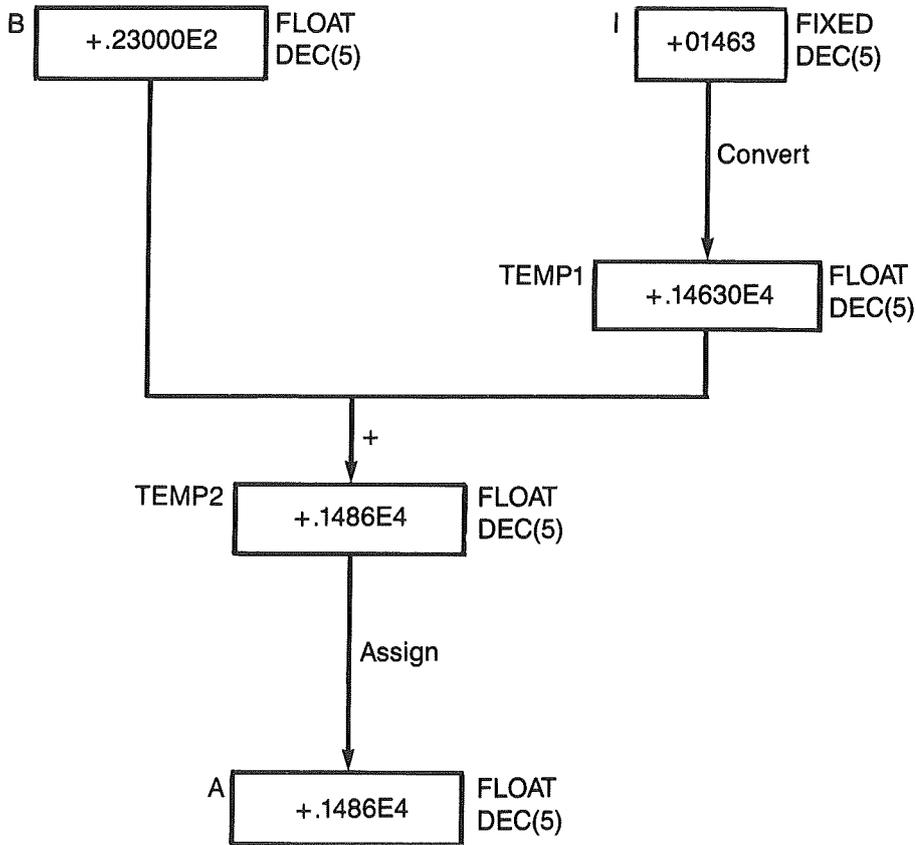
As another example, consider the following program segment

```
DECLARE (A,B) FLOAT DECIMAL(5);
DECLARE I FIXED DECIMAL(5);
..
B = 23;
I = 1463;
A = B + I;
```

Note the last assignment statement. This example is quite different from the preceding one because a FIXED variable is added to a FLOAT variable. Because a FIXED quantity cannot be directly added to a FLOAT quantity, PL/I performs an implicit conversion. The value of I is converted to FLOAT, and that result is added to B.

Figure 6-2 illustrates how PL/I executes this assignment statement. As this figure shows, the value of I is converted to FLOAT DECIMAL(5), and the result of the conversion is stored in an intermediate target called TEMP1. Then, B is added to TEMP1, with the result stored in a new intermediate target called TEMP2. This result is then assigned to A.

The following pages contain the rules for determining whether an implicit conversion is required and what the data types of intermediate targets are.



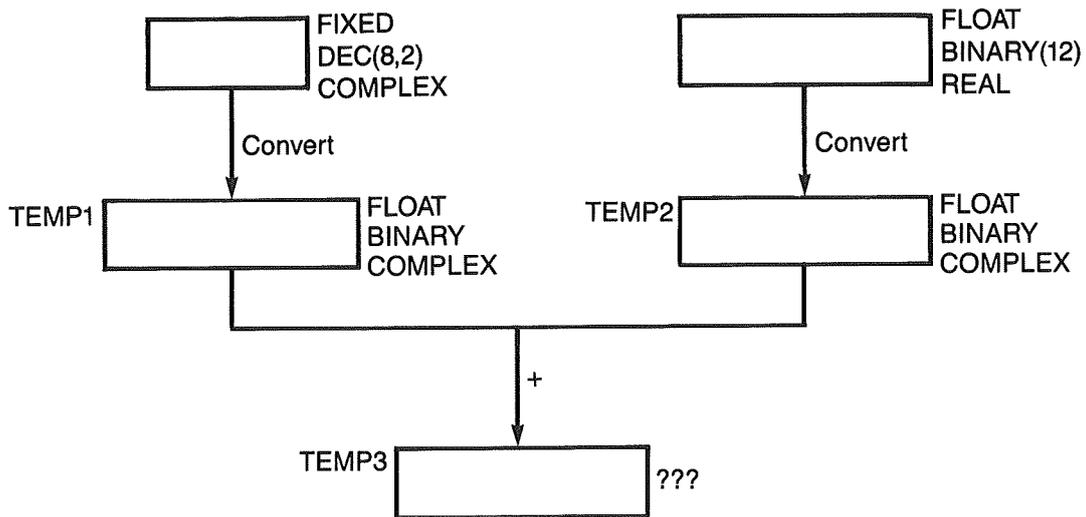
Implicit Conversions
Figure 6-2

Derived Common Base, Scale, and Mode

The last example showed that when a FLOAT operand is added to a FIXED operand, PL/I converts the FIXED operand to FLOAT before performing the addition operation.

Figure 6-3 illustrates what happens when a program adds two operands, one of which is FIXED DECIMAL(8,2) COMPLEX and the other FLOAT BINARY(12) REAL. This example is more complicated than the preceding one because the two operands differ not only in the scale (one is FIXED and the other is FLOAT), but also in the base (one is DECIMAL and the other is BINARY) and the mode (one is COMPLEX and the other is REAL). Figure 6-3 shows that PL/I performs two implicit conversions. Each of the two operands is converted to the derived common base, scale, and mode, with the appropriate converted precisions, and the results are stored in the two intermediate targets, TEMP1 and TEMP2. This section defines derived common base, scale and mode. The next section defines

converted precision. These concepts make it possible to derive the complete data types of TEMP1 and TEMP2. A later section specifies the rules for determining the complete data type for the result of the addition, stored in target TEMP3.



Conversion of Scale, Base, and Mode
Figure 6-3

Derived Common Scale: Table 6-4 gives the rules for defining the derived common scale for two operands. Informally, one can say that FLOAT is a higher data type than FIXED. More precisely, if either of the operands is FLOAT, the derived common scale is FLOAT; otherwise, the derived common scale is FIXED.

For example, consider the following program segment:

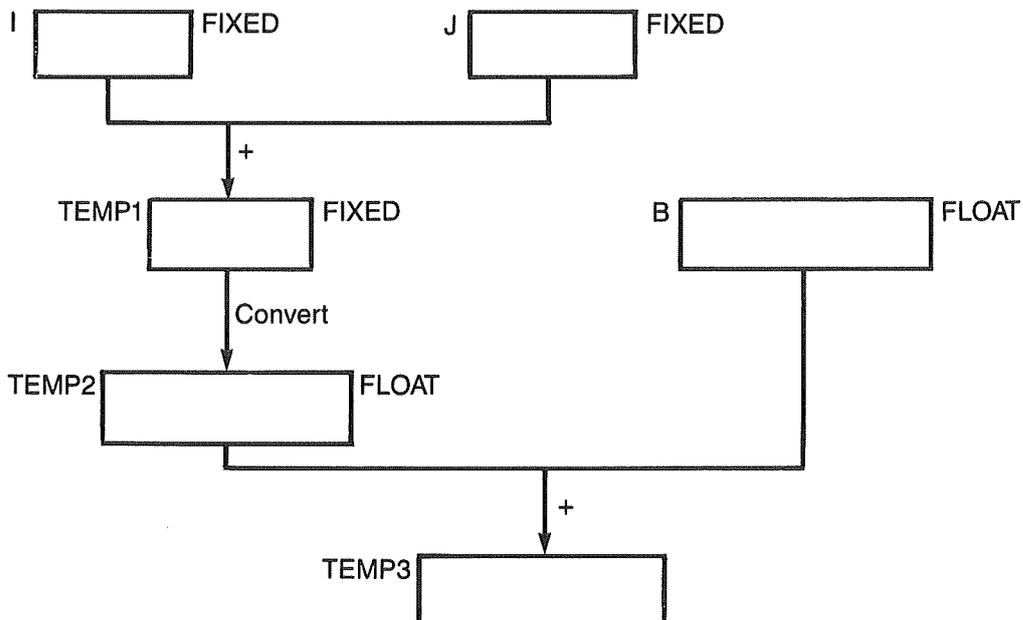
```

DECLARE (A,B) FLOAT;
DECLARE (I,J) FIXED;
..
A = I + J + B;
  
```

The assignment statement contains two addition operations, and, by the priority rules, the + on the left is computed before the + on the right. The result is illustrated in Figure 6-4. As this figure shows, the value of I + J is computed as a FIXED quantity, and the result is stored in TEMP1. That quantity is then converted to FLOAT, and the results stored in TEMP2, so that it can be added to the value of B, which is FLOAT.

Table 6-4
Derived Common Scale for Two Operands

Operand 1	Operand 2		
	Arithmetic or pictured-numeric with scale=FLOAT	Arithmetic or pictured-numeric with scale=FIXED	Not either arithmetic or pictured-numeric
Arithmetic or pictured-numeric with scale=FLOAT	FLOAT	FLOAT	FLOAT
Arithmetic or pictured-numeric with scale=FIXED	FLOAT	FIXED	FIXED
Not either arithmetic or pictured-numeric	FLOAT	FIXED	FIXED



Derived Common Scale
Figure 6-4

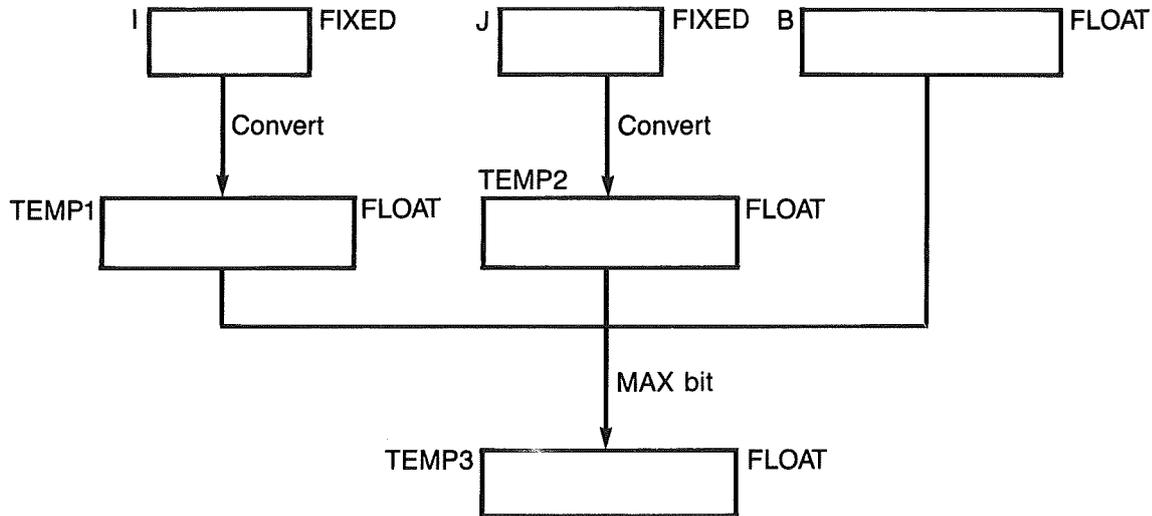
PL/I Reference Guide

Note that PL/I executes this assignment statement differently from the way other languages, such as FORTRAN, execute it. The FORTRAN language would not perform any additions until all the operands had been converted to floating point. This means that the value of I would be converted to floating point, the value of J would be converted to floating point, and then those two quantities would be added together to the value of B to get a floating point result. PL/I, by contrast, postpones all conversions as long as possible, performing only those that are explicitly dictated by the operator rules.

Nonetheless, there are circumstances when more than two operands must be converted to a common derived scale. Although this cannot happen with any of the ordinary operators, it can happen with the MAX built-in function:

```
DECLARE (A,B) FLOAT;  
DECLARE (I,J) FIXED;  
..  
A = MAX(I,J,B);
```

This program segment is quite similar to the preceding one, in that the final assignment statement assigns to A some operation performed on the three variables I, J, and B. In the last example, the operation was addition, which is performed on operands two at a time. In that case, the value I + J was computed first without any regard for the data type of B. In the current example, the MAX built-in function looks at all three operands at once. This means that all three operands must be converted to the common derived scale before any further progress can be made. The result is illustrated in Figure 6-5. The value of I is converted to FLOAT, and the result is stored in TEMP1. Similarly, the value of J is converted to FLOAT and stored in TEMP2. At that point, all three operands to MAX are FLOAT, and so the computation of which number is the maximum can take place, with the results stored in TEMP3.



Derived Common Scale — Three Options
Figure 6-5

In cases such as this, use a more complete rule for the derived common scale of two operands: given two or more operands, the derived common scale for the operands is **FLOAT** if at least one of the operands is either arithmetic or pictured-numeric with a scale of **FLOAT**; otherwise, the derived common scale is **FIXED**.

Derived Common Mode: The rule for the derived common mode for two operands can be stated informally as follows: the **COMPLEX** data type is higher than the **REAL** data type. The precise rule is given in Table 6-5.

Table 6-5
Derived Common Mode for Two Operands

Operand 1	Operand 2		
	Arithmetic or pictured-numeric with mode=COMPLEX	Arithmetic or pictured-numeric with mode=REAL	Not either arithmetic or pictured-numeric
Arithmetic or pictured- numeric with mode=COMPLEX	COMPLEX	COMPLEX	COMPLEX
Arithmetic or pictured- numeric with mode=REAL	COMPLEX	REAL	REAL
Not either arithmetic or pictured- numeric	COMPLEX	REAL	REAL

Certain built-in functions, such as MAX, require the computation of the derived common mode for more than two operands. The complete rule, then, is as follows: given two or more operands, the derived common mode for the operands is COMPLEX if at least one of the operands is either arithmetic or pictured-numeric with a mode of COMPLEX; otherwise, the derived common mode is REAL.

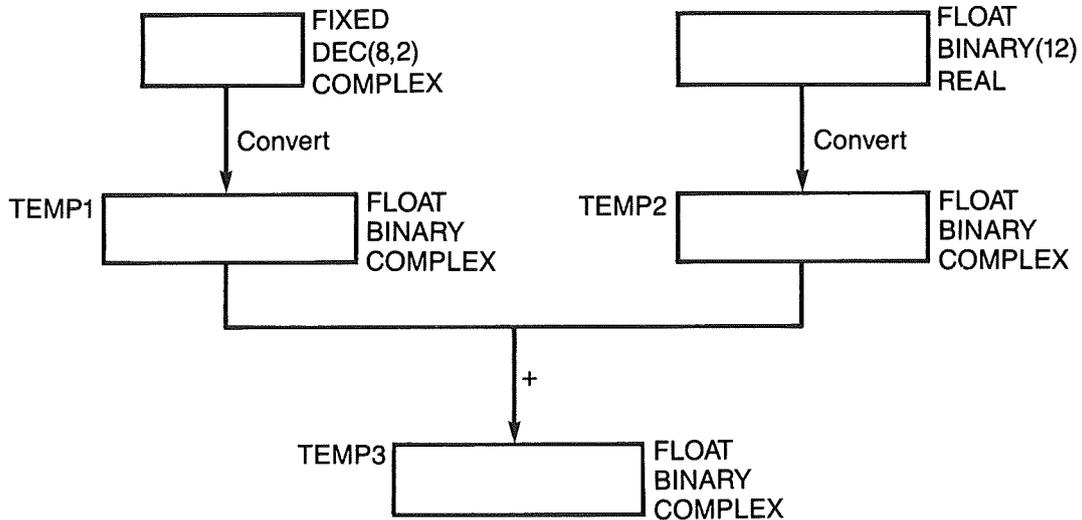
Derived Common Base: We may summarize the rule for deriving the common base of two operands by saying that BINARY is a higher data type than DECIMAL, but there is a slight additional complication. It is possible that one of the operands is a BIT string, and for the purposes of determining the derived common base, a BIT string is considered to be BINARY. The precise rule is given in Table 6-6.

Table 6-6
Derived Common Base for Two Operands

Operand 1	Operand 2		
	Arithmetic or base=BINARY	BIT string	Neither BIT string nor BINARY arithmetic
Arithmetic with base= BINARY	BINARY	BINARY	BINARY
BIT string	BINARY	BINARY	BINARY
Neither BIT string nor BINARY arithmetic	BINARY	BINARY	DECIMAL

As in the case of scale and mode, certain built-in functions require the derived common base for more than two operands simultaneously. The complete rule, then, is as follows: given two or more operands, the derived common base is BINARY if at least one of the operands either is a BIT string or is arithmetic with a base of BINARY; otherwise, the base is DECIMAL.

Refer to Figure 6-6. If the two operands are FIXED DECIMAL COMPLEX and FLOAT BINARY REAL, the derived common scale, base, and mode are FLOAT BINARY COMPLEX. As the figure shows, each of the operands must be converted to the derived common scale, base, and mode before addition can take place. This figure does not yet indicate the precision of TEMP1 or TEMP2. The subject of converted precision will be treated in the next section.



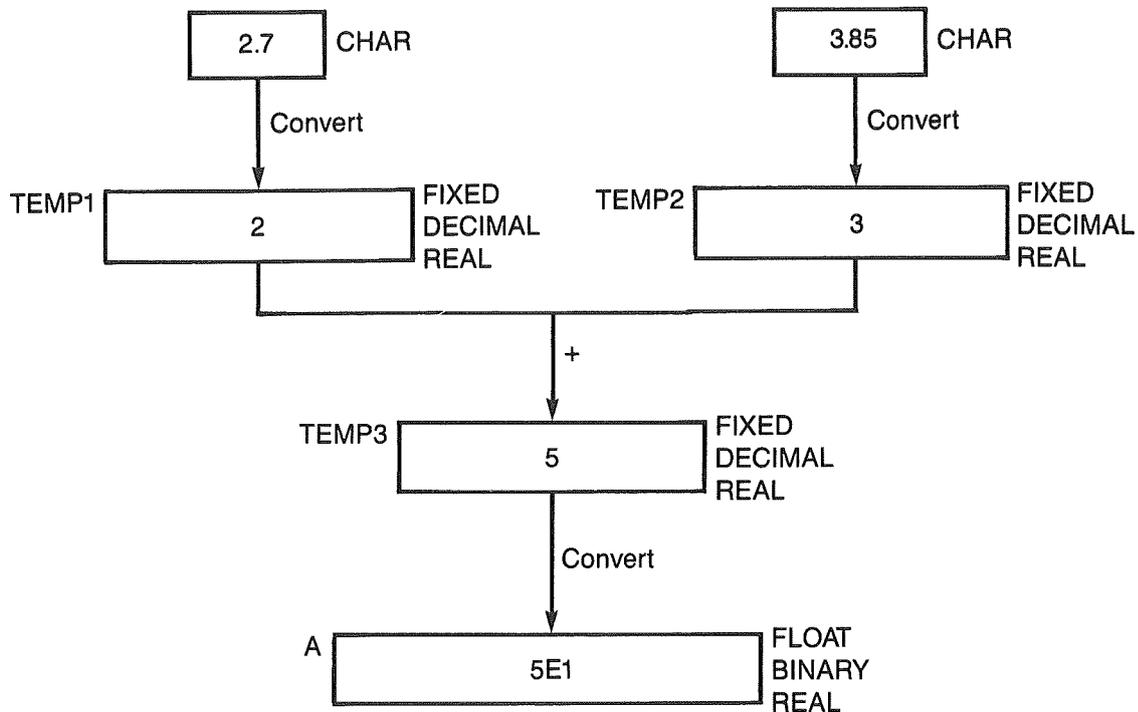
Derived Common Scale, Base, and Mode
Figure 6-6

As noted above, one of the operands could be a BIT string. Actually, PL/I permits you to use operands of any computational data type in arithmetic expressions. If you use a string, PL/I must convert that string to the appropriate arithmetic data type, with the derived common base, scale, and mode. You are strongly urged to avoid such implicit conversions from string to arithmetic. As an example of the trouble you can get into, consider the following program segment:

```

DECLARE A FLOAT;
..
A = '2.7' + '3.85';
  
```

In the assignment statement, you are adding together two CHARACTER string values, presumably expecting A to be assigned the value 6.55. Actually, something quite different happens, as illustrated in Figure 6-7. Since PL/I must add together two CHARACTER values, the derived common base, scale, and mode are FIXED DECIMAL REAL, with a scale factor of zero. The result is that each of the CHARACTER values is converted to an integer before the addition takes place, and so A is assigned the value 5.



Conversion of CHARACTER Values
Figure 6-7

Converted Precision

This section deals with the precision and scale factor of the intermediate target created as the result of an implicit conversion during the evaluation of an expression.

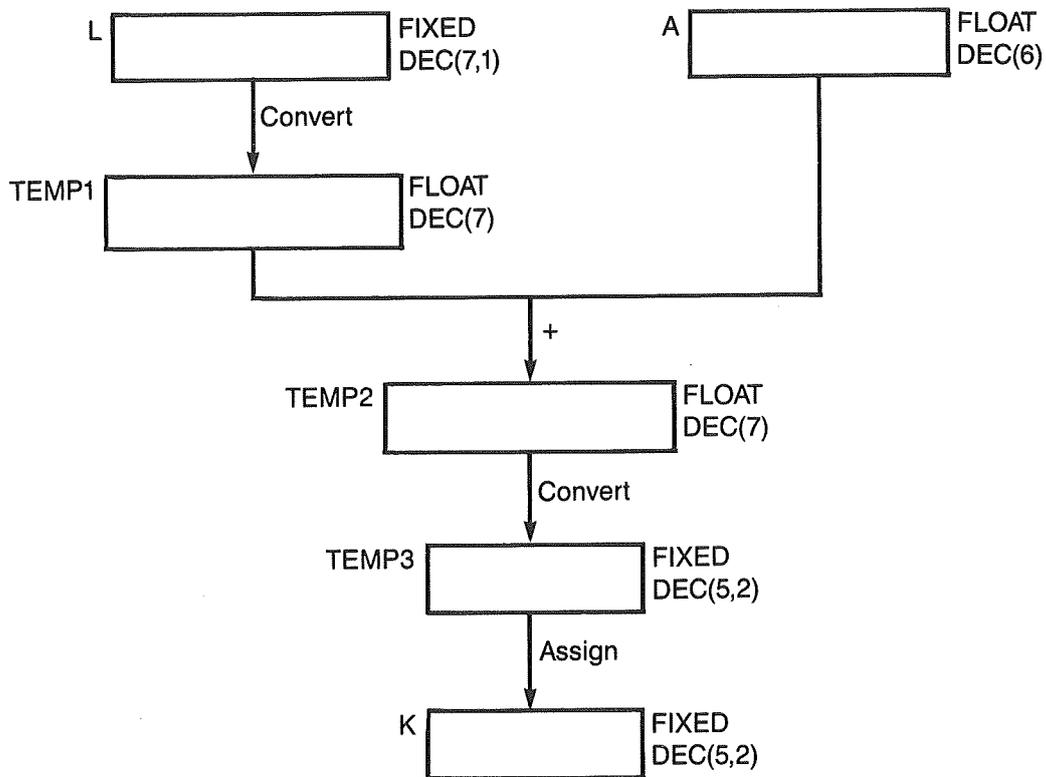
Consider the following program segment:

```

DECLARE K FIXED DECIMAL(5,2);
DECLARE L FIXED DECIMAL(7,1);
DECLARE A FLOAT DECIMAL(6);
..
K = L + A;
  
```

The assignment statement in the last line adds together a FIXED and a FLOAT value, and so an implicit conversion is required. Figure 6-8 illustrates the result. (In this figure, no mode is shown for any of the data types, because the converted precision does not depend upon the mode.) The derived common scale and base of the data types for L

and A are FLOAT DECIMAL. Therefore, PL/I converts the value of L to FLOAT DECIMAL, and stores the result in a target called TEMP1. Since the value of L contains seven digits, PL/I also gives TEMP1 a precision of 7, since no additional digits are necessary. The target precision of 7 is the converted precision for the conversion of FIXED DECIMAL(7,1) to FLOAT DECIMAL. Figure 6-8 also shows the precisions of all temporary targets. The precision of TEMP2 is determined by the rules for addition, which are discussed later in this chapter.



Converted Precision
Figure 6-8

The precision and scale factor of TEMP3 are determined by the precision of the target variable K. For this reason, this precision does not follow the rules for the converted precision described in this section, because the conversion is considered explicit rather than implicit.

Table 6-7 spells out the rules for converted precisions in implicit conversion in general. The values of the converted precisions depend only upon the scale and base of the source and target data types; they do not depend upon the mode of the source and target data types. This table shows, for each combination of source scale and base with target scale and base, the formulas for determining the converted precision.

Table 6-7
Converted Precisions in Implicit Conversions

Target Data Type	Source Data Type			
	FIXED BINARY (p,q)	FIXED DECIMAL (p,q)	FLOAT BINARY (p)	FLOAT DECIMAL (p)
FIXED BINARY (r,s)	r=p s=q	r=MIN(31, CEIL(p*3.32)+1) s=CEIL(q*3.32)	No implicit conversion possible	No implicit conversion possible
FIXED DECIMAL (r,s)	r=CEIL(p/3.32)+1 s=CEIL(q/3.32)	r=p s=q	No implicit conversion possible	No implicit conversion possible
FLOAT BINARY (r)	r=p	r=CEIL(p*3.32)	r=p	r=CEIL(p*3.32)
FLOAT DECIMAL (r)	r=CEIL(p/3.32)+1	r=p	r=MIN(14, CEIL(p/3.32))	r=p

These notes refer to Table 6-7:

- In converting from FIXED to FLOAT, the converted precision of the FLOAT target equals the number of digits in the FIXED source. The reason is that in FLOAT targets we are interested only in the appropriate number of significant digits.
- In the PL/I language, it is never possible to have an implicit conversion from FLOAT to FIXED; FLOAT to FIXED conversions must be explicit. That is why no formula is given in those four positions in Table 6-7.
- In going from DECIMAL to BINARY, or vice versa, the constant 3.32 is used. This constant is approximately equal to the common logarithm (that is, the logarithm to the base 10) of 2. This constant is chosen because it is possible to prove mathematically that if you represent a large integer in both BINARY and DECIMAL, it will require approximately 3.32 times as many digits to represent it in BINARY as it does in DECIMAL. In the table, the use of the function CEIL is made in order to indicate that the result of multiplying or dividing by 3.32 should be rounded up to the next higher integer.
- In two cases, using the straightforward formula for the converted precision would result in a precision that is larger than the maximum permitted for that scale and base on Prime

equipment. In those two cases, the MIN built-in function is used in order to indicate that, if the formula results in a precision that is larger than the maximum allowed, the maximum precision should be used.

Derived Common String Type

This section deals with operations performed on BIT or CHARACTER string data. If you perform a string operation on two operands with different string data types, PL/I must determine the derived common string type for the two operands, and then convert the operands to that data type, if necessary.

Table 6-8 gives the rules for the derived common string type. For example, if you have a statement containing an expression that concatenates two operands, PL/I uses this table to determine whether to do a BIT string concatenation or a CHARACTER string concatenation.

Table 6-8
Derived Common String Type for Two Operands

Operand 1	Operand 2		
	BIT string	CHARACTER string	Not a string
BIT string	BIT	CHARACTER	CHARACTER
CHARACTER string	CHARACTER	CHARACTER	CHARACTER
Not a string	CHARACTER	CHARACTER	CHARACTER

The rule can be specified as follows: given two operands, if both of them are BIT string operands, the derived common string type is BIT; otherwise, the derived common string type is CHARACTER.

PL/I EXPRESSION OPERATORS

This section defines all of the PL/I operators used in the evaluation of expressions. While you are reading this section, keep in mind the following general rules:

- Each operator has either one or two operands. If it has two operands, it is called an infix operator. If it has only one operand, it is called a prefix operator.
- PL/I evaluates an operator by creating a target of an appropriate data type and then storing the result of the operation in that target.
- At the time that PL/I is compiling your program, it determines the data type of the target. The data type of the target depends only on the operator and on the data types of the operands; the data type of the target does not depend upon the value of the operands, even when the value is known at compile time. (There is an exception to this rule in certain cases of the ** operator.)
- While the data type of the target is determined at compile time, the value of the target is determined when the program executes.

Infix + and - Operators

These operators perform ordinary arithmetic addition and subtraction. Given the operands \underline{x} and \underline{y} , PL/I evaluates $x + y$ or $x - y$ according to the following rules:

1. PL/I determines the derived common scale, base, and mode of the data types for \underline{x} and \underline{y} .
2. PL/I converts \underline{x} to the data type of the derived common base, scale, and mode, with the appropriate converted precision. PL/I does the same for \underline{y} . These conversions can take place in either order.
3. PL/I creates a target having a data type with the derived common base, scale, and mode, and a precision as defined by Table 6-9.
4. PL/I performs the addition or subtraction of the operand values, and stores the result in the target.

Table 6-9

Precision of Target Results for
Addition or Subtraction of Two Numbers

Derived Common Scale & Base of x and y	Converted Precision of x	Converted Precision of y	Precision of Target for x+y or x-y
FIXED BINARY	(p,q)	(r,s)	(m,n) where m=MIN(31,MAX(p-q,r-s)+ MAX(q,s)+1) n=MAX(q,s)
FIXED DECIMAL	(p,q)	(r,s)	(m,n) where m=MIN(14,MAX(p-q,r-s)+ MAX(q,s)+1) n=MAX(q,s)
FLOAT BINARY	(p)	(r)	(m) where m=MAX(p,r)
FLOAT DECIMAL	(p)	(r)	(m) where m=MAX(p,r)

In order to understand Table 6-9, consider the following example. Suppose x has a data type of FIXED DECIMAL(2) and y has a data type of FIXED DECIMAL(3). Then the maximum value that x can have is 99, and the maximum value that y can have is 999. Therefore, the maximum value of $x + y$ is $99 + 999$, or 1098. Since this result contains four digits, PL/I creates a target of FIXED DECIMAL(4).

When a scale factor is involved, the reasoning is similar. For example, suppose x has a data type of FIXED DECIMAL(2) and y has a data type of FIXED DECIMAL(3,1). Then the maximum value of x is 99, and the maximum value of y is 99.9, and so the maximum value of $x + y$ is 198.9. Since the maximum result contains four digits, with one digit to the right of the decimal point, the data type of the target is FIXED DECIMAL(4,1).

Occasionally the desired target precision is larger than the maximum precision supported by PL/I. For example, if x and y are both FIXED DECIMAL(14), the desired target precision is FIXED DECIMAL(15). Unfortunately, this exceeds the maximum precision allowed, and so PL/I uses a target of FIXED DECIMAL(14).

In the table, the rules for FLOAT BINARY and FLOAT DECIMAL follow a general rule for FLOAT that holds for all the arithmetic operators. The precision of the target is the maximum of the precision of the operands. In practical terms, this means that if both operands are single precision, the target is single precision; if at least one of the operands is double precision, the target is double precision.

Infix * Operator

The asterisk (*) is used for multiplication. Given the two operands x and y, PL/I evaluates $x * y$ as follows:

1. PL/I determines the derived common base, scale, and mode for the data types for x and y.
2. PL/I converts x to the data type of the derived common base, scale, and mode, with the appropriate converted precision. PL/I does the same for y. These conversions can take place in either order.
3. PL/I creates a target having a data type with the derived common base, scale, and mode, and a precision as defined by Table 6-10.
4. PL/I performs the multiplication operation, storing the result into this target.

Table 6-10
Precision of Target Results for Multiplication of Two Numbers

Derived Common Scale, Base of <u>x</u> and <u>y</u>	Converted Precision of <u>x</u>	Converted Precision of <u>y</u>	Precision of Target for $x*y$
FIXED BINARY	(p,q)	(r,s)	(m,n) where $m = \min(31, p+r+1)$ $n = q+s$
FIXED DECIMAL	(p,q)	(r,s)	(m,n) where $m = \min(14, p+r+1)$ $n = q+s$
FLOAT BINARY	(p)	(r)	(m) where $m = \max(p, r)$
FLOAT DECIMAL	(p)	(r)	(m) where $m = \max(p, r)$

In Table 6-10, the reasoning is similar to the reasoning in the case of addition and subtraction, although there is a slight additional complication.

For example, suppose x has a data type of FIXED DECIMAL(2) and y has a data type of FIXED DECIMAL(3). Then the largest value of x is 99 and the largest value of y is 999. Therefore, the maximum value that $x * y$

can have is 98901, which contains five digits. This would seem to indicate that PL/I should create a target having the data type FIXED DECIMAL(5). However, the precisions defined in Table 6-10 must work whether the mode is REAL or COMPLEX. If \underline{x} were FIXED DECIMAL(2) COMPLEX and \underline{y} were FIXED DECIMAL(3) COMPLEX, then $x * y$ could have six digits in the real part or the imaginary part of the result. For this reason, PL/I adds 1 to the precision that you would expect, and the target is FIXED DECIMAL(6).

The reasoning in the other cases follows the reasoning for addition and subtraction.

Infix / Operator

The slash (/) is used to perform arithmetic division. Given two operands, \underline{x} and \underline{y} , PL/I evaluates x / y as follows:

1. PL/I determines the derived common base, scale, and mode of the data types for \underline{x} and \underline{y} .
2. PL/I converts \underline{x} to the data type of the derived common base, scale, and mode, with the appropriate converted precision. PL/I does the same for \underline{y} . These conversions can take place in either order.
3. PL/I creates a target having a data type with the derived common base, scale, and mode, and a precision as defined in Table 6-11.
4. PL/I performs the division operation, and stores the result into this target.

Table 6-11 contains a fundamental difference from the tables for addition, subtraction, and multiplication. The difference is in the FIXED case, where the precision of the target is always the maximum allowed, irrespective of the precisions of the operands. The reason is that division can produce approximate results, even when the operands are exact. PL/I attempts to preserve as much accuracy as possible by creating a target with a large number of digits to the right of the decimal point, thus preserving as much accuracy in the quotient as possible.

Table 6-11
Precision of Target Results for Division

Derived Common Scale Base of x and y	Converted Precision of x	Converted Precision of y	Precision of Target x/y
FIXED BINARY	(p,q)	(r,s)	(m,n) where $m=31$ $n=31-p+q-s$
FIXED DECIMAL	(p,q)	(r,s)	(m,n) where $m=14$ $n=14-p+q-s$
FLOAT BINARY	(p)	(r)	(m) where $m=\max(p,r)$
FLOAT DECIMAL	(p)	(r)	(m) where $m=\max(p,r)$

For example, suppose x and y are each FIXED DECIMAL(1). This means that x and y each contain only one decimal digit, and so x / y can have at most one digit to the left of the decimal point. Since PL/I can support a maximum precision of fourteen decimal digits, and since at most one digit can appear to the left of the decimal point, PL/I provides a target with thirteen digits to the right of the decimal point. Therefore, the target has size FIXED DECIMAL(14, 13).

Similar reasoning is involved when x and y have nonzero scale factors. In addition, the rules for FLOAT are similar to the rules for addition, subtraction, and multiplication.

Be aware that the use of FIXED division can get you into unexpected trouble. Consider, for example, the following statement:

$$A = 25 + 1/3;$$

You may be surprised to learn that this statement cannot execute properly according to the rules for PL/I. To understand why, consider the data types of the temporaries used in the evaluation of the expression $25 + 1/3$.

Since 1 and 3 are single decimal digits, they have a data type of FIXED DECIMAL(1). Therefore, the target for the division operation will have a data type of FIXED DECIMAL(14,13), as described in the preceding paragraph. When the division operation is performed, the result stored in the target will be 0.333333333333. An examination of Table 6-9

will reveal that, since 25 has a data type of FIXED DECIMAL(2), the target for the addition operation is FIXED DECIMAL(14,13), which allows only one digit to the left of the decimal point. Therefore, the target is not large enough to hold the value of the expression.

The reader should note that this is not a bug in the PL/I compiler, but rather is a consequence of the precisely defined rules of the PL/I language. Many mathematicians have examined these rules to try to eliminate this fixed division problem. However, it has turned out that any change to the rules has simply moved the problem somewhere else, and so no satisfactory change has been found.

To the programmer who finds this situation disconcerting, remember the following: never do FIXED division. If you wish to divide two FIXED quantities, convert one of them to FLOAT before doing the division. Of course, if you must maintain the accuracy provided by FIXED division, by all means do it, but be certain that you check the sizes of the targets in the expression, and make sure that they will be large enough to hold all the intermediate results.

Infix ** Operator

The exponentiation operator, the double asterisk (**), is handled quite differently from the other arithmetic operators because of some of its unique mathematical properties. For example, PL/I will sometimes handle a case like x^{**2} in a special way, because this equals $x * x$. This means, for example, that in certain cases PL/I will use the fact that the value of the constant in the exponent is 2 to define the data type of the target of the operation. This is the only case where PL/I uses the value of an operand to determine the data type of the target of the operand.

Special cases: Let \underline{x} and \underline{k} be the two operands of **, such that \underline{k} is an integer constant or variable; that is, such that the derived scale and mode of \underline{k} are FIXED REAL, with a scale factor of zero. Then the cases are as follows:

1. If \underline{k} is a positive integer constant, and the derived scale and base of \underline{x} are FIXED BINARY, and the converted precision of \underline{x} is (p,q) , such that $(p + 1) * k - 1 \leq 31$, then the target data type is FIXED BINARY(m,n), where $m = (p + 1) * k - 1$, and $n = q * k$.
2. If \underline{k} is a positive integer constant, and the derived scale and base of \underline{x} are FIXED DECIMAL, and the converted precision of \underline{x} is (p,q) , such that $(p + 1) * k - 1 \leq 14$, then the target data type is FIXED DECIMAL(m,n) where $m = (p + 1) * k - 1$, and $n = q * k$.
3. If neither of the above two cases holds, PL/I proceeds as follows: it converts \underline{x} to the derived base and mode of \underline{x} , and a scale of FLOAT, with the appropriate converted precision.

The target of the ** operation will have the same data type as the converted value of x.

The reasoning behind these special cases is that if x is FIXED with a small precision, and n is a fairly small constant, then the exponentiation operation can be performed using ordinary FIXED multiplication, and the result stored into a FIXED target. On the other hand, if k is an integer value, but is either a large constant or a variable, then the exponentiation operation can still be performed by repeated multiplication, but the multiplication must be done using a scale of FLOAT, to prevent a result that is too large for the target.

General case: If none of the special cases apply, the general rule applies. Let x and y be the two operands. Then the general rule for the evaluation of $x^{**}y$ is as follows:

1. PL/I determines the derived common base and mode of the data types of x and y. PL/I does not determine the derived common scale, because FLOAT will be used.
2. PL/I converts x to the data type of the derived common base and mode with FLOAT, and with the appropriate converted precision. PL/I does the same for y. These conversions can take place in either order.
3. PL/I creates a target having a data type with the derived common base and mode, with a scale of FLOAT, and with a precision equal to the maximum of the converted precisions of x and y.
4. PL/I performs the exponentiation operation, storing the result into the target.

Infix || Operator

The double vertical bar (||) is the concatenation operator. On Prime terminals, it may be entered as || or !!. It is applied to two string values, and the result of concatenating two string values is simply to stick the strings together. Concatenation may apply to either CHARACTER or BIT string values.

Let x and y be the two operands of the concatenation operation. Then PL/I determines the value of $x || y$ as follows:

1. PL/I determines the derived common string type of the data types for x and y. PL/I converts x to the derived common string type. PL/I does the same for y. These conversions can take place in either order.
2. PL/I creates a target having a data type with the derived common string type.

3. PL/I concatenates the two string values, storing the result in the target.

Notice an interesting difference between the ways PL/I handles numeric targets and string targets. The PL/I rules specify that the precision of numeric targets must always be known at compile time. However, the length of a string target need not be known until the program actually executes.

Infix Comparison Operators

PL/I provides eight comparison operators. These operators are listed in Table 6-12.

You may use any of the eight comparison operators to compare two REAL numeric values or two string values. In addition, you may use either of the first two (= or ^=) to compare values of all other data types (COMPLEX numeric or noncomputational). Use of any of the last six comparison operators with any of these latter data types is illegal.

The target for any of the comparison operators is always BIT(1). If the comparison is true, the value of the result is '1'B. If the result of the comparison is false, the result value is '0'B.

Table 6-12
Comparison Operators

Operator	Meaning
=	Equals
^=	Does not equal
<	Is less than
<=	Is less than or equal
>	Is greater than
>=	Is greater than or equal
^<	Is not less than
^>	Is not greater than

When the two operands of the comparison operator have different data types, PL/I must convert the operands to the same data type. The conversion rules are as follows:

1. If at least one of the two operands is numeric or pictured-numeric, a numeric comparison is done. The rules are described below.

2. If both of the operands are either string or pictured-character, a string comparison is done. The rules are described below.
3. If one of the operands has a noncomputational data type, the other operand must have the same noncomputational data type. The only exception is that a POINTER value may be compared with an OFFSET value. For all noncomputational data type comparisons, only the operators = and ^= may be used.

Numeric comparisons: Let x and y be the two operands being compared. PL/I compares them as follows:

1. PL/I determines the derived common base, scale, and mode of the data types of x and y .
2. PL/I converts x to the data type of the derived common base, scale, and mode, with the appropriate converted precision. PL/I does the same for y . These conversions can take place in either order.
3. PL/I creates a BIT(1) target.
4. PL/I makes the appropriate comparison, and sets the target to the '0'B if the result is false, and to '1'B if the result is true. If the derived common mode of x and y is COMPLEX, only the comparisons = and ^= are permitted.

String comparisons: Let x and y be the two operands being compared. PL/I compares them as follows:

1. PL/I determines the derived common string type of the data types for x and y .
2. PL/I converts x to the derived common string type. PL/I does the same for y . These conversions can take place in either order.
3. If the resulting strings do not have the same length, PL/I pads the shorter one on the right, so that they do have the same length. If the derived string type is CHARACTER, PL/I pads the shorter string on the right with blanks. If the derived common string type is BIT, PL/I pads the shorter string on the right with 0-bits.
4. PL/I creates a BIT(1) target. PL/I uses the rules in the following paragraphs to determine whether the two strings are equal or whether one is greater than the other.
5. If the two strings of equal length are identical, they are considered to be equal. Otherwise, to determine which is larger, PL/I compares the two strings character by character or bit by bit, moving from left to right in the two strings, until it finds a pair of characters or bits that are unequal.

6. If the derived string type is CHARACTER, the two unequal characters are compared in the ASCII collating sequence. If the first character comes before the second character in this collating sequence, the first string is considered smaller than the second string; otherwise, the second string is considered smaller.
7. If the derived string type is BIT, one of the unequal bits must be a 0-bit, and the other must be a 1-bit. The string that contains the 0-bit is considered smaller than the other string.

To illustrate the rule for CHARACTER string comparisons, take the strings 'ANT' and 'ANVIL'. PL/I pads the shorter string with blanks to the length of the longer string, with the result that the first string becomes 'ANTbb'. PL/I then compares the two strings character by character as follows:

```

A N T b b
↑ ↑ ↑
A N V I L
    
```

The first pair of unequal characters are T and V. Since T comes before V in the ASCII collating sequence, the first string is considered smaller than the second. See Appendix B for the ASCII Collating Sequence.

As a second example, suppose PL/I is comparing the strings 'ANT' and 'ANTELOPE'. PL/I pads the shorter of these strings with blanks, and then compares the results character by character, as follows:

```

A N T b b b b b
↑ ↑ ↑
A N T E L O P E
    
```

The first pair of unequal characters is found in fourth position. These characters are a blank and E. Since the blank comes before E in the ASCII collating sequence, PL/I considers the first string to be smaller than the second.

Infix &, |, and ! Operators

The ampersand (&), vertical bar (|), and exclamation point (!) are the logical operators. The symbol & represents AND; | and ! both stand for OR. Normally, you think of these operators as simple logical connectors, as in the statement:

```
IF A > 3 & B = 5 THEN STOP;
```

In reality, PL/I treats &, |, and ! as operators on BIT strings. That is, the operands of these operators must be BIT strings, and the results are BIT strings.

Suppose x and y are two operands. Then PL/I computes $x \& y$ or $x | y$ as follows:

1. PL/I converts x to BIT. PL/I does the same for y. These conversions can occur in either order.
2. If these two BIT strings do not have the same length, PL/I pads the shorter string with 0-bits, so that it is the same length as the longer string.
3. PL/I creates a target having a data type of BIT and a length equal to the common length of the two strings.
4. Moving from left to right, PL/I takes bits from corresponding bit positions in the two operands, and produces a result bit from these two operand bits. If the operator is &, the result bit is shown in Table 6-13.

Table 6-13
Bit Results from &

		Operand 1	
		0 B	1 B
Operand 2	0 B	0 B	0 B
	1 B	0 B	1 B

If the operator is |, the result bit is shown in Table 6-14. PL/I stores the result bit into the corresponding bit position of the target.

Table 6-14
Bit Results from |

		Operand 1	
		0 B	1 B
Operand 2	0 B	0 B	1 B
	1 B	1 B	1 B

Prefix + and - Operators

The minus sign, the prefix - operator, takes one operand and negates it by changing a positive number to negative or a negative number to positive. The plus sign, the prefix + operator, also takes one operand, but it has no effect whatsoever on the value of that operand except to force a conversion to numeric, if the data type is not already numeric.

Given an operand x, PL/I evaluates +x or -x as follows:

1. PL/I determines the derived base, scale, and mode of the data type for x. Since there is only one operand, the derived base, scale, and mode are the same as the base, scale, and mode of x, unless x is not numeric.
2. PL/I converts x to the data type of the derived base, scale, and mode with the appropriate converted precision. Notice that no conversion is ever necessary if x is already numeric.
3. PL/I creates a target having a data type with the derived base, scale, and mode, and a precision as defined by Table 6-15.
4. PL/I evaluates +x by simply using the converted value of x. PL/I evaluates -x by reversing the sign of the converted value of x.

Notice that in Table 6-15, the precision of the target for +x or -x is the same as the data type for the converted value of x.

Table 6-15
Precision of Target for Unary Plus and Minus Operators

Derived Scale and Base of x	Converted Precision	Precision of Target for $+x$ or $-x$
FIXED BINARY	(p, q)	(m, n) where $m=p$ $n=q$
FIXED DECIMAL	(p, q)	(m, n) where $m=p$ $n=q$
FLOAT BINARY	(p)	(m) where $m=p$
FLOAT DECIMAL	(p)	(m) where $m=p$

Prefix ^ Operator

The caret (^) is the logical operator corresponding to NOT. It is a prefix operator taking only one operand. That operand must be a BIT string, and the result of the operation is BIT.

Given an operand x , PL/I computes \hat{x} as follows:

1. PL/I converts x to BIT.
2. PL/I creates a BIT target that is the same length as the string just discussed.
3. Proceeding from left to right, PL/I fetches a bit from the operand, uses that bit to compute a result bit as shown in Table 6-16, and stores the result bit in the corresponding position of the result BIT string.

Table 6-16
Bit Results from ^

Operand	Result
'0'B	'1'B
'1'B	'0'B

SCALAR CONVERSION RULES FOR COMPUTATIONAL DATA TYPES

This section contains the rules for conversion from one computational data type to another -- numeric to numeric, string to numeric, and numeric to string.

Most conversions among noncomputational data types are illegal in PL/I. For example, it is illegal to convert POINTER to FIXED, or FORMAT to LABEL. The one exception is that, under certain circumstances, it is legal to convert POINTER to OFFSET, or vice versa. These conversions are discussed in Chapter 7.

Numeric To Numeric Conversions

If a data value is converted from one numeric or pictured-numeric data type to a different one, there may be a change in value. Such a conversion would involve a change in one or more of the base, scale, mode, and precision data attributes.

Change in base: If there is a change in base from BINARY to DECIMAL, or vice versa, there is usually a round-off error in converting noninteger values. Notice, however, that if the scale is FLOAT, there is no difference in the internal representation of FLOAT DECIMAL and FLOAT BINARY on Prime computers.

Change in scale: If there is a change in scale from FLOAT to FIXED, truncation occurs if the FLOAT value contains more digits to the right of the decimal or binary point than are provided for by the scale factor of the FIXED data type. A conversion from FIXED DECIMAL or FIXED BINARY to FLOAT usually has a round-off error, because, on Prime equipment, FLOAT is always represented internally in binary. PL/I performs a mode conversion from REAL to COMPLEX by using an imaginary part of 0. PL/I converts COMPLEX to REAL by discarding the imaginary part of the COMPLEX value.

Change of precision: In the case of FLOAT, an increase in precision will mean no change in value, while a decrease in precision will result in either no change in value or a truncation of significant digits in going from an internal representation of double precision FLOAT to single precision.

How a change in precision for FIXED data values works depends upon whether digits are added to or removed from the left end of the number or from the right end, and upon the position of the decimal point as determined by the scale factors of the source and target data types. If digits are added to either end, there is no change in value, because PL/I does the conversion by filling the new positions with zeros. If digits are removed from the right-hand end of the source value, truncation takes place. If digits are removed from the left-hand end of the source value, and if the values of these digits are nonzero, then a SIZE error occurs.

Numeric To CHARACTER Conversions

Since the rules are fairly complicated, it is not recommended to most programmers to use numeric to CHARACTER conversions. Programmers who need to convert numeric values to CHARACTER values should do so by means of PICTURE variables, which permit precise specification of the CHARACTER string.

When you convert a numeric value to CHARACTER, PL/I represents the numeric value as a DECIMAL constant, where the constant has the data type of the numeric variable, and stores the constant in the CHARACTER string, usually with leading blanks. The precise rules are listed in the following paragraphs.

If the base of the numeric value is BINARY, PL/I converts the value to DECIMAL, with the appropriate converted precision. PL/I then proceeds with the DECIMAL value, as described below.

If the value is FIXED DECIMAL(p,q) REAL, where q is greater than or equal to zero and p is greater than or equal to q , then PL/I forms a CHARACTER string containing $p+6$ characters. There are three cases:

1. If q equals zero, the CHARACTER string contains up to p digits, with a leading minus sign for negative numbers and a leading space for positive numbers. There is no decimal point in the string.
2. If q is greater than zero and p equals q , PL/I forms a string of the form 'SO.DDD...D', where the S is replaced by a blank if the number is positive and by a minus sign if the number is negative, and there are q digits following the decimal point.
3. If q is greater than zero and p is greater than q , PL/I forms a constant containing at most $p-q$ digits before the decimal point, preceded by a minus sign for negative numbers with q digits following the decimal point. There will always be at least one leading blank.

If the numeric value is FIXED DECIMAL(p,q) REAL, where either q is greater than p or q is less than zero, then PL/I forms a CHARACTER string of the form 'Sddd...dFS(n)', where there are p digits and no decimal point, and where the n is replaced by an integer constant with as many digits as necessary to represent the position of the implied decimal point.

If the numeric value is FLOAT DECIMAL(p) REAL, PL/I forms a character string of the form 'Sd.ddd...dESdd', where there are $p-1$ digits following the decimal point, and there is a two-digit signed characteristic. For double-precision numbers, the form is 'Sd.ddd...dESdddd'. The length of this CHARACTER string is $p+6$ for single precision, and $p+8$ for double precision.

If the numeric value is COMPLEX, and if n is the length of the character string for the corresponding REAL data type, then PL/I forms a CHARACTER string of length $2n + 1$, formed as follows:

1. Get the CHARACTER string representations of the real and imaginary parts of the COMPLEX value. If the imaginary part is positive, replace the last leading blank in the corresponding CHARACTER string with a +.
2. Concatenate the two CHARACTER strings together, and concatenate an I to the result.
3. Rearrange the characters in this string by moving all leading blanks to the left-hand side.

CHARACTER to Numeric

A CHARACTER to numeric conversion is legal if the CHARACTER string contains a legal PL/I numeric constant, possibly with leading and trailing blanks. The CHARACTER string may contain a COMPLEX constant, consisting of a real and an imaginary part added together, with the letter I following the imaginary part. In no case may the constant contain any embedded blanks.

If the CHARACTER string contains only blanks, or is the null string, the conversion is still legal, and the corresponding numeric value is 0.

Numeric to BIT

This conversion is quite complicated and is not recommended. The detailed rules are described in the following paragraphs.

If the data type of the numeric value is COMPLEX, take only the real part of the numeric value. If the numeric value is negative, take its absolute value.

The length of the BIT string will be p , where p depends upon the data type of the numeric value, as follows:

1. If the numeric value is FIXED BINARY (r,s), p equals $r - s$.
2. If the numeric value is FIXED DECIMAL (r,s), p equals $\text{CEIL}(3.32 * (r - s))$.
3. If the numeric value is FLOAT BINARY (r), p equals r .
4. If the numeric value is FLOAT DECIMAL (r), p equals $\text{CEIL}(3.32 * r)$.

In all four cases, if the resulting value of p is negative, p equals 0. If $p > 31$, let p equal 31.

Convert the numeric value to the data type `FIXED BINARY(p,0) REAL`, to obtain a nonnegative integer. Represent this integer as a binary number containing p bits, and those bits will form the resulting `BIT` string.

BIT to Numeric

When converting `BIT` to numeric, PL/I examines the `BIT` string and treats the bits as an unsigned binary integer. This integer is evaluated and converted to `FIXED BINARY(31) REAL`. This numeric value is then converted to the appropriate data type.

BIT to CHARACTER

PL/I converts a `BIT` string to a `CHARACTER` string of the same length. Each 0-bit in the `BIT` string is replaced with the character '0' in the `CHARACTER` string, and each 1-bit in the `BIT` string is replaced with the character '1' in the `CHARACTER` string. For example, '1011'B would be converted to '1011'.

CHARACTER to BIT

This conversion is legal only if the `CHARACTER` string contains only the characters '0' and '1'. PL/I translates the `CHARACTER` string to a `BIT` string of the same length by replacing the character '1' with a 1-bit, and the character '0' with a 0-bit.

Conversions Involving PICTURE Variables

If a variable is pictured-string, its conversions are treated the same as for `CHARACTER`.

When converting a value to a pictured-numeric data type, PL/I converts the source value to numeric, and then forms the string value of the `PICTURE` variable by performing the `PICTURE` editing rules. This is true even if the source value is a `CHARACTER` string.

In conversions from a pictured-numeric value to a `CHARACTER` value, the string value of the `PICTURE` variable is used. For converting to any other data type, the numeric value of the `PICTURE` variable is used.

Special Conversion Built-in Functions

PL/I follows the conversion rules given in this chapter to determine what data attributes and precisions should apply to conversion targets.

PL/I provides a number of built-in functions that allow you to specify explicit conversions. For example, consider the following statements:

```
DECLARE A FLOAT DECIMAL(7);
DECLARE B FIXED DECIMAL(5);
C = A + B;
```

According to the rules, PL/I computes $A + B$ by converting B to `FLOAT DECIMAL(5)` and then performing a `FLOAT` addition.

However, suppose you would prefer that PL/I convert A to `FIXED`, and then compute $A + B$ using `FIXED` addition. You might do something like the following:

```
C = FIXED(A, 9, 2) + B;
```

PL/I executes this statement by converting A to the data type `FIXED DECIMAL(9,2)` and then adding that result, using `FIXED` addition rules, to the value of the variable B .

As another example, suppose the assignment statement is changed to

```
C = A + FLOAT(B, 15);
```

PL/I converts B to `FLOAT DECIMAL(15)` rather than `FLOAT DECIMAL(5)` before performing the `FLOAT` addition operation.

`FIXED` and `FLOAT` are examples of PL/I built-in functions that allow you to specify explicitly what kinds of conversions PL/I should make in evaluating your expression. Other built-in functions provided for this purpose are `BINARY`, `DECIMAL`, `REAL`, `COMPLEX`, `PRECISION`, `CHARACTER`, and `BIT`. These functions are fully described in Chapter 14.

In addition, PL/I allows you to specify the precision of the targets of numeric operations. For example, using the same declarations as in the preceding example, PL/I evaluates $A + B$ in the statement

```
C = A + B;
```

by creating a target with the attributes `FLOAT DECIMAL(7)`. To specify a target with a different precision attribute, use the `ADD` built-in function, as follows:

```
C = ADD(A, B, 12);
```

PL/I computes the value of `A + B` by creating a target with the attributes `FLOAT DECIMAL(12)`.

In addition to `ADD`, you may use `SUBTRACT`, `MULTIPLY`, and `DIVIDE` to specify the target precision attributes for a numeric operation. These functions are fully defined in Chapter 14.

AGGREGATE TARGETS AND PROMOTION

Expressions considered so far have involved only scalar elements. This section deals with PL/I expressions involving aggregates, specifically arrays, structures, and arrays of structures.

Aggregate Expressions without Promotions

Consider the following program segment:

```
DECLARE A(4);
DECLARE B(4) INITIAL(1,3,8,2);
DECLARE C(4) INITIAL(2,5,6,7);
..
A = B + C;
```

In this assignment statement, the array `B` is added to the array `C` with the results stored in the array `A`. When you add two arrays together, the result is an array value. The array value is computed by adding corresponding elements of the two arrays being added.

In the above assignment statement, `B + C` is computed to form the array `(3,8,14,9)`. This array value is then assigned to the array `A`.

In an assignment statement involving arrays, PL/I requires that all arrays must have the same number of dimensions, and the same upper and lower bounds.

Similarly, it is possible to have structure expressions. Consider the following program segment:

```
DECLARE 1 S, 2 A, 2 B;  
DECLARE 1 T, 2 A INITIAL(5), 2 B INITIAL(6);  
DECLARE 1 U, 2 A INITIAL(4), 2 B INITIAL(2);  
..  
S = T + U;
```

Here, the assignment statement adds together two like structures. The result is that the corresponding structure members are added together to form a structure target. In this case, the structure target will contain two members, and the values will be 9 and 8. This structure target is then assigned to the structure S, with the result that S.A will have the value 9, and S.B will have the value 8.

These aggregate operations may be used with any infix or prefix operators, or with any of the PL/I built-in functions. However, notice that when you multiply two arrays together, you do not get what is normally referred to as matrix multiplication. All operations are performed by performing the operation on the individual corresponding scalar members.

Scalar to Array Promotions

When scalars and arrays are involved in an expression or in an assignment statement, it is often necessary to promote or convert each scalar to a corresponding array. Consider the following program segment:

```
DECLARE A(4);  
A = 5;
```

In this assignment statement, a scalar constant, 5, is being assigned to an array A. In order for PL/I to make this assignment, it must promote the scalar 5 to the array (5,5,5,5). PL/I can then assign this array to A.

Now consider the following program segment:

```
DECLARE A(4);  
DECLARE B(4) INITIAL(2,3,4,5);  
X = 4;  
A = B * X;
```

The last assignment statement contains the expression $B * X$, which multiplies the array B by the scalar X .

In order for this expression to be evaluated, the scalar X must be promoted to an array. Since X has the value 4, PL/I promotes X to the array (4,4,4,4). PL/I then multiplies these two arrays to get a new array value, (8,12,16,20). This array value is then assigned to A .

Note

These scalar-to-array promotion rules were affected by the 1976 changes in the ANS PL/I standard. Earlier implementations of PL/I (for instance, IBM PL/I) may give unexpected answers to program segments like the following:

```
DECLARE A(4) INITIAL(1,2,3,4);
A = A(2) * A;
```

By the current rules, the scalar value $A(2)$ is multiplied by the array A , and the result is assigned back to A . As we have described, $A(2)$ is promoted to the temporary array (2,2,2,2). This temporary array then multiplies A to form a new array, (2,4,6,8).

On older compilers, however, this assignment statement is performed differently. In fact, the assignment statement is equivalent to the following three statements:

```
DO K = 1 TO 4;
A(K) = A(2) * A(K);
END;
```

As you can verify, the results of this assignment would be (2,4,12,16).

Scalar to Structure Promotions

Just as you can promote a scalar to an array, you can promote a scalar to a structure. Consider the following example:

```
DECLARE 1 S, 2 A, 2 B;
S = 7;
```

In this example, the scalar constant 7 is promoted to a structure containing two elements, each of whose values is 7. This structure is then assigned to S, with the result that S.A = 7 and S.B = 7.

Next, consider the following example:

```

DECLARE 1 S, 2 A, 2 B;
DECLARE 1 T, 2 A INITIAL(4), 2 B INITIAL(5);
X = 7;
S = T * X;
    
```

To execute the last assignment statement, the scalar X is promoted to a structure containing two elements, with the respective values 7 and 7. This structure is multiplied by the structure T, and the result is assigned to S, with the result that S.A equals 28 and S.B equals 35.

Promotion to Array of Structures

It is possible for any aggregate type, a scalar, an array, or a structure, to be promoted to an array of structures. Consider the following example:

```

DECLARE 1S(4), 2A, 2B;
DECLARE 1C(4) INITIAL(5,6,7,8);
DECLARE 1T, 2D INITIAL(2), 2E INITIAL(3);
..
S = C + T + 4;
    
```

Whenever a structure and an array are added together, PL/I promotes each of them to an array of structures. In this last assignment statement, the array C, the structure T, and the scalar 4 are each promoted to an array of structures, where the array contains four members and each of these members is a structure containing two scalar values. You may picture these promotions as resulting in the following intermediate array of structure values:

<u>C</u>	<u>T</u>	<u>4</u>
5 5	2 3	4 4
6 6	2 3	4 4
7 7	2 3	4 4
8 8	2 3	4 4

EVALUATING EXPRESSIONS

These three arrays of structures are added together to get a new array of structures that may be pictured as follows:

11 12

12 13

13 14

14 15

These values are then assigned to the array of structures S.

7

Storage Management

Whenever you use a computer system, you are limited as to the amount of storage you may use. When you write a PL/I program, your program uses some storage for the program instructions and some storage for the data used by the program. This chapter deals with techniques for managing areas occupied by the data of your program, covering the following topics:

- The use of PL/I storage types to control the allocation, initialization, and freeing of data storage. If you have a program that is too large to run efficiently, you may be able to use this information to reduce the amount of storage required by your data. The technique for storage reduction is to control the allocation of large blocks of data, so that each large block occupies storage only while it is needed. This section also covers list processing techniques.
- Techniques for overlaying storage. It is possible for two variables to share the same storage area, or for one variable to share a portion of the storage area of another variable. In these cases, the storage of one variable is said to overlay the storage of another variable. This section gives several techniques for overlaying storage.
- Extent expressions and the INITIAL attribute. Extent expressions and INITIAL attributes appear in declarations of variables. An extent expression is an expression defining an array bound or string length. An INITIAL attribute specifies how a variable is to be initialized. This section defines these

terms more fully and discusses how to use variables in these options in order to have variable-sized arrays and strings.

- The `EXTERNAL` and `INTERNAL` scope attributes. This section describes how to use the `EXTERNAL` attribute to permit one external procedure to access the data areas that are used by a second external procedure.
- Named constants. Certain PL/I constants have names. For example, a statement label is a named constant. This section explains how to specify named constants, and how the corresponding noncomputational variable data types can be declared.

TYPES OF STORAGE

In the PL/I language, the terms storage type and storage class usually refer to the manner in which the storage is to be allocated, initialized, and freed according to the specifications of your program.

The term allocation refers to the operation of associating a specific block of storage or memory with a variable. For example, suppose your program contains the following declaration:

```
DECLARE A(1000);
```

This declaration specifies that A is to be an array that occupies 1000 words of storage. You may not use the array A until PL/I has allocated a specific storage block of 1000 words, and associated that block with A. Normally, PL/I does this automatically without your even realizing it. However, PL/I provides several optional techniques that give you complete control over when the storage block is allocated.

The term initialization refers to the operation of assigning an initial value to a variable, by means of the `INITIAL` attribute in the `DECLARE` statement. The `INITIAL` attribute is discussed more fully later in this chapter. For now, look at the following example:

```
DECLARE RANGE FLOAT INITIAL(5.3);
```

This statement specifies that `RANGE` is a `FLOAT` variable, and that PL/I is to give `RANGE` an initial value of 5.3. Initialization takes place just after allocation. If allocation takes place more than once for a given variable, initialization takes place the same number of times.

The term freeing refers to the operation of releasing a block of storage that has been allocated and used by some variable. Under certain circumstances, if you have finished using a particular

variable, you may release the storage occupied by that variable, so that other variables can use that same storage area. If you do this carefully, you may be able to keep your program size smaller.

The following paragraphs list the types of storage supported by PL/I and summarize the rules for when storage of each type is allocated, initialized, and freed.

1. If a variable has the `STATIC` storage class, it is allocated and initialized when your program begins executing, and it is freed when your program finishes executing. This is conceptually the simplest storage class to understand, since you may assume that the storage is always available.
2. The default storage type is `AUTOMATIC`. `AUTOMATIC` is the same as `STATIC` for variables that are declared in your main procedure. However, if an `AUTOMATIC` variable is declared in a subroutine or function procedure, or within any `BEGIN` block or `ON` unit, PL/I allocates and initializes the storage when the block is invoked, and frees the storage when the block is terminated. If the block is invoked and terminated several times during execution of a program, the storage is allocated and freed each time.
3. If a variable has the `CONTROLLED` storage class, allocation, initialization, and freeing of the storage are done entirely under your control. When your program executes an `ALLOCATE` statement, PL/I allocates and initializes the storage for the `CONTROLLED` variable. When your program executes a `FREE` statement, PL/I frees the storage. No other allocation or freeing operations take place except that, of course, when your program ends, all storage is released.
4. The `BASED` storage class gives you the most control of all over your storage management. As with `CONTROLLED`, `BASED` storage is allocated and initialized only with an explicit `ALLOCATE` statement, and freed with an explicit `FREE` statement. However, when used with `POINTER` variables, `BASED` storage provides a very powerful capability for list processing and other applications where you must handle sophisticated data structures in your program.
5. A variable with the `PARAMETER` storage type is one that is specified in a `PROCEDURE` statement. The allocation, initialization, and freeing operations that we have discussed do not really apply to `PARAMETER` variables, because such a variable is simply a pointer back to the argument that was specified when the procedure was called or referenced. This is discussed more fully in Chapter 8 on procedures.
6. A `DEFINED` variable is one that does not have its own storage; instead, it shares storage with a variable of a different storage type. `DEFINED` variables are described later in the section TECHNIQUES FOR OVERLAYING STORAGE.

7. A temporary variable is one that is invisible to you, the user, but that PL/I requires in order to execute your program. For example, when your program evaluates an expression, PL/I must often allocate a storage area for intermediate results. This is described in Chapter 6. PL/I allocates such temporary storage when your program requires it, and frees that storage when your program is finished with it. The programmer has no direct control over allocation or freeing of temporary storage.

In PL/I terminology, the term storage type refers to any of the seven classifications described in the preceding paragraphs. The term storage class is more restrictive, referring to those types described in the first four of these paragraphs, namely `STATIC`, `AUTOMATIC`, `CONTROLLED`, and `BASED`. The remainder of this chapter concentrates on the four storage classes.

STATIC and AUTOMATIC Storage

The most elementary storage class is the `STATIC` storage class. Suppose your program contains the following declaration:

```
DECLARE A FIXED STATIC INITIAL(0);
```

This declaration specifies that A is to be a `FIXED` variable, with the `STATIC` storage class attribute, and is to be initialized to the value zero. When your program begins execution, PL/I allocates the variable A and initializes it to 0. The storage for A remains allocated until your program terminates. This is true no matter where this declaration appears in your program, even if it appears in a subroutine or function procedure.

If you declare a variable and do not specify any storage class, PL/I supplies the `AUTOMATIC` storage class by default. Consider the following declaration in which `AUTOMATIC` is explicitly declared:

```
DECLARE B FIXED AUTOMATIC INITIAL(0);
```

This declaration is the same as that for A given above, except that now the storage class is `AUTOMATIC` rather than `STATIC`. If this declaration appears in the `MAIN` procedure, and not within any internal `PROCEDURE` or `BEGIN` block, the result is the same as for `STATIC`; that is, when your program begins execution, B is allocated and initialized to 0, and is freed when your program terminates. On the other hand, if this declaration appears within a subroutine or function procedure, or within a `BEGIN` block, B is allocated and initialized each time the block is invoked, and is freed each time the block is terminated.

To understand more fully the differences between `STATIC` and `AUTOMATIC`, consider the next example. This program contains declarations for four variables, `A`, `B`, `X`, and `Y`.

```

P:   PROC OPTIONS(MAIN);
      DECLARE A FIXED STATIC INIT(0);
      DECLARE B FIXED AUTOMATIC INIT(0);
      PUT LIST(A,B);
      ...
      CALL Q;
      ...
      CALL Q;
      ...
Q:   PROC;
      DECLARE X FIXED STATIC INIT(5);
      DECLARE Y FIXED AUTOMATIC INIT(5);
      PUT LIST(X,Y);
      X, Y = 10;
      RETURN;
      END Q;
      END P;

```

The first two declarations are the same as those already discussed. Since these declarations are in the main procedure, and are not inside any internal `PROCEDURE` or `BEGIN` block, the effects of the `STATIC` and `AUTOMATIC` storage classes are essentially identical. Both `A` and `B` are allocated and initialized to 0 when the program begins execution and are freed when the program ends execution. The `PUT` statement in the fourth line of the program prints the value 0 twice.

Note the declarations of `X` and `Y` in the same example. These declarations appear inside the internal procedure `Q`. Because `X` has the storage class `STATIC`, it is allocated and initialized only once, at the time the program begins execution. This means that when procedure `Q` is called the first time, `X` has the value 5. The `PUT` statement prints a value of `X` equal to 5. Notice that the assignment statement on the following line assigns the value 10 to `X`. This means that when procedure `Q` is called the second time, `X` has the value 10, and so the value 10 is printed by the `PUT` statement inside procedure `Q`, the second time `Q` is called.

On the other hand, because `Y` is `AUTOMATIC` (the default), it is allocated and initialized each time procedure `Q` is called, and is freed each time procedure `Q` is terminated. This means that when the value 10 is assigned to `Y` the first time `Q` is called, that value is lost as soon as `Q` is terminated, since `Y` is freed at that point. The second time `Q` is called, `Y` is reallocated and reinitialized to the value 5, so that the `PUT` statement prints a value of 5 for `Y`. In summary, then, the first time `Q` is called, the `PUT` statement inside `Q` prints the values 5 and 5. The second time `Q` is called, the same `PUT` statement prints the values 10 and 5.

AUTOMATIC Storage in a Recursive Procedure

When a declaration for an AUTOMATIC variable appears inside a recursive procedure, it is possible to have several different allocations of the variable in existence at once. Consider the example below. The procedure Q is recursive, and it contains declarations for a STATIC variable X and an AUTOMATIC variable Y. When Q calls itself, there are two simultaneous active invocations of Q. Since X is static, there is only one allocation of X, and all invocations of Q refers to the same storage area for X.

```

P:   PROC OPTIONS(MAIN);
      ...
      CALL Q;
Q:   PROC RECURSIVE;
      DCL X FIXED STATIC INIT(5);
      DCL Y FIXED AUTOMATIC INIT(5);

      X = X + 1;
      Y = Y + 1;
      ...
      CALL Q;
      ...
      END Q;
      END P;

```

On the other hand, Y is AUTOMATIC. If there are two or more active invocations of Q at one time, there are an equal number of allocations of Y in existence at the same time. This means that each invocation of Q references a different allocation of Y.

The multiple allocation of an AUTOMATIC variable in a recursive procedure is very much like a stack mechanism, since each invocation of the procedure may reference only the most recent allocation of the AUTOMATIC variable. CONTROLLED storage also implements a form of stack mechanism.

CONTROLLED Storage Class

If you declare a variable to have the CONTROLLED storage class, you have complete control over the allocation, initialization, and freeing operations for the storage for that variable. For example, suppose your program contains the declaration

```
DECLARE MAT(100,100) FLOAT CONTROLLED;
```

Then PL/I allocates no storage for MAT, and it is illegal to reference MAT until you specify that storage is to be allocated. Use the statement

```
ALLOCATE MAT;
```

to specify that PL/I is to allocate storage for MAT. (If the declaration for MAT contained an INITIAL attribute, PL/I would also perform initialization, right after allocation.) Each ALLOCATE statement can specify a maximum of one segment of storage. In the above example, if MAT required more than one segment, you would have to break it into several storage modules and use a separate ALLOCATE statement for each one.

After your program has completed processing MAT, release the storage for other use by executing the statement

```
FREE MAT;
```

This statement says that the allocation for MAT is to be released.

Use the ALLOCATE statement to create multiple allocations of a CONTROLLED variable. When you do this, a reference to the CONTROLLED variable becomes a reference to the most recent allocation. Therefore, PL/I supports multiple allocations of CONTROLLED storage as a stack mechanism. The built-in function ALLOCATION takes a CONTROLLED variable as an argument and returns the number of allocations of that CONTROLLED variable currently in existence.

Consider, for example, the following:

```
DECLARE C FIXED CONTROLLED;
ALLOCATE C;
C = 5;
ALLOCATE C;
C = 10;
PUT SKIP LIST(ALLOCATION(C), C);
FREE C;
PUT SKIP LIST(ALLOCATION(C), C);
FREE C;
PUT SKIP LIST(ALLOCATION(C));
```

This program segment prints three lines of output. The program segment begins by creating two allocations of C, setting the first one to 5 and the second one to 10. Therefore, the first PUT statement prints the values 2 and 10, since there are currently two allocations of C in existence, and the most recent allocation has a value of 10. The second PUT statement appears after a FREE statement. This FREE statement frees the second allocation of C, so that the second PUT statement prints the values 1 and 5. The second FREE statement leaves no remaining allocations of C, with the result that the last PUT statement prints the value 0.

One of the most useful features of CONTROLLED storage is that it allows you to have arrays with variable-sized dimension bounds and strings with variable-sized maximum lengths. Variable extent expressions are discussed later in this chapter.

BASED Storage Class and POINTER Variables

BASED storage is the most elementary of the storage classes because it reduces storage management to a set of operations that put everything under the control of the programmer. On the other hand, it is the most sophisticated of the storage classes since by using it the programmer can implement very advanced list processing applications.

As in the case of CONTROLLED storage, you must allocate BASED storage by means of an explicit ALLOCATE statement. However, in the case of CONTROLLED storage, multiple allocations are handled by PL/I by means of a stack mechanism, which makes only the most recent allocation available to the programmer. Furthermore, PL/I maintains total control over the locations of CONTROLLED storage.

In the case of BASED storage, it is the programmer who must write the program so as to keep track of the location of each allocation of storage. If there are several allocations in existence at the same time, the programmer may reference any of the allocations at any time. Furthermore, the programmer may free the allocations in any order.

The user keeps track of the location of each allocation by means of the POINTER variable. POINTER is an example of a noncomputational data type of PL/I, so called because you may not perform ordinary arithmetic or string operations on it. A POINTER value is, conceptually, the storage address of an allocation of storage. When you allocate a BASED variable, you specify, in the ALLOCATE statement, the name of a POINTER variable that PL/I will set to point to the block of storage being allocated. If later you wish to reference that block of storage, do so by means of the same POINTER value.

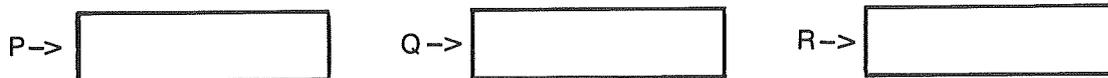
To illustrate these concepts, consider the following program segment:

```
DECLARE X FIXED BASED;  
DECLARE (P,Q,R) POINTER;  
ALLOCATE X SET(P);  
ALLOCATE X SET(Q);  
ALLOCATE X SET(R);  
P->X = 5;  
Q->X = P->X + 1;  
R->X = P->X + Q->X;
```

In this example, the variable X is BASED. We have also declared three pointer variables, P, Q, and R, allowing us to keep track of three different allocations of the BASED variable X. (Note that POINTER is a

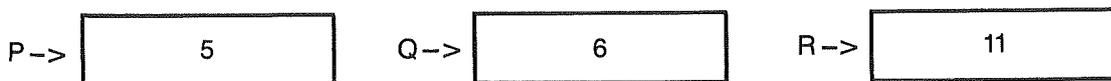
data type, while `BASED` is a storage class. `P`, `Q`, and `R` all have a storage class of `AUTOMATIC`, the default storage class.)

The program segment above contains three `ALLOCATE` statements. Each of these statements causes PL/I to allocate a block of storage for `X`. In each case, an appropriate `POINTER` variable `P`, `Q`, or `R`, as specified in the `ALLOCATE` statement, is set by PL/I to point to the region just allocated. The result of these three allocations is shown in Figure 7-1. As this figure shows, there are three blocks of storage, pointed to respectively by `P`, `Q`, and `R`.



Pointer Variables
Figure 7-1

By using the appropriate `POINTER` variable, you may reference any of three allocations at any time. To do this, use the right arrow symbol (`->`). Precede this symbol with the appropriate `POINTER` variable, and follow this symbol with the `BASED` variable. Therefore, to reference the first of three allocations in the example above, use `P->X`; to reference the second, `Q->X`; and to reference the third, use `R->X`. The last three assignment statements in the program segment above use these references to assign values to these three allocations. The resulting values are shown in Figure 7-2.



Assignment With Pointer Variables
Figure 7-2

Use the same symbol to free any of the allocations. For example, if you wish to free the second allocation of the three in the example above, use the statement

```
FREE Q->X;
```

Once this statement has been executed, the POINTER variable Q no longer has a valid value.

Another way to understand BASED storage is to compare it to STATIC storage. Suppose Y is declared as follows:

```
DECLARE Y FIXED STATIC;
```

Then the variables X and Y have the same data type, FIXED, but have different storage classes. If you reference the variable Y in an expression, PL/I automatically knows from just the identifier Y both what the data type is and where the storage is. However, this is not the case with the BASED variable X. If you use X in an expression, PL/I does not have enough information to get a value, since a BASED variable has a data type but has no storage location. In order to provide PL/I with a storage location, you must also use a POINTER variable such as P->X. The P portion specifies the location of the data, and the X portion specifies the data type of the value.

You may, if you wish, specify a default POINTER variable in your DECLARE statement for a BASED variable. For example, the declaration of X could be written

```
DECLARE X FIXED BASED(P);
```

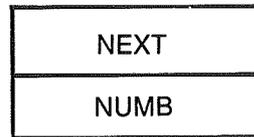
This declaration for X specifies a default POINTER variable of P. This means that if you reference X with no POINTER qualifier, PL/I assumes that you mean a POINTER variable of P. You may still specify any POINTER variable you wish with X simply by using the -> operator.

A List Processing Example

The following short example of list processing by means of a linked list illustrates BASED storage and POINTER variables. The basic declarations of our program example are

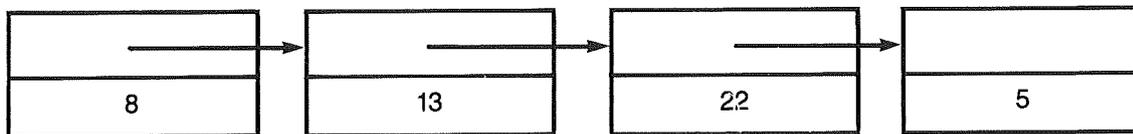
```
DECLARE 1 REC BASED,  
       2 NEXT POINTER,  
       2 NUMB FIXED;  
DECLARE (FIRST, P, Q) POINTER;
```

In the first declaration, REC is a BASED structure, with two scalar members NEXT and NUMB. This means that you may think of an allocation of REC as looking something like Figure 7-3. As that figure shows, an allocation of REC is a block of storage containing two values, NEXT and NUMB.



A BASED Structure
Figure 7-3

A linked list is a collection of allocations of REC, with each block in the list pointing to the next block in the list. That is, each allocation of REC contains a POINTER value and a FIXED value. The POINTER value can point to another allocation of REC. For example, Figure 7-4 shows four allocations of REC, with each pointing to the next one in the list. The result is that this is a linked list of four numbers, 8, 13, 22, and 5.



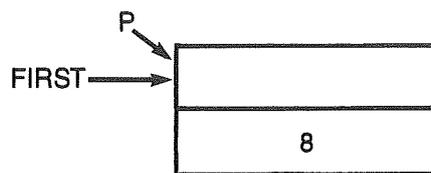
A Linked List
Figure 7-4

The following PL/I code segment creates a linked list from 50 values that are input using GET LIST:

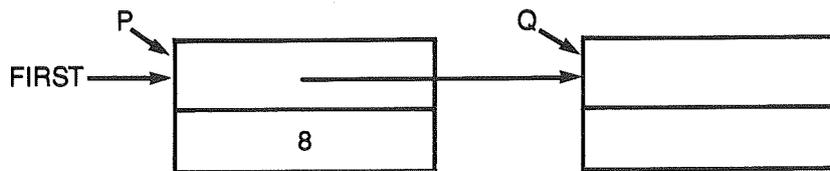
```

ALLOCATE REC SET(FIRST);
GET LIST(FIRST->NUMB);
P = FIRST;
DO K = 2 TO 50;
  ALLOCATE REC SET(Q);
  P->NEXT = Q;
  GET LIST(Q->NUMB);
  P = Q;
END;
P->NEXT = NULL();
  
```

The first statement allocates a storage block for REC, and sets the pointer variable FIRST to point to that storage block. The second statement inputs a data value, and stores it in the NUMB portion of the newly allocated REC storage block. Notice that FIRST->NUMB is a shorthand notation for FIRST->REC.NUMB. The third statement is an assignment statement, assigning the POINTER value FIRST to the POINTER variable P. The result is that P and FIRST now point to the same storage block. If we assume that the first input value is 8, the situation after the first three lines of the above code may be pictured as follows:

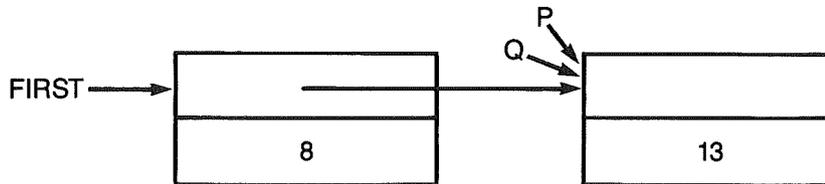


The next six lines of code form a DO group that allocates the next 49 blocks in the linked list, and inputs the 49 data values to be stored in the 49 storage blocks. Let us examine what happens during the first iteration of this loop. The statement ALLOCATE REC SET(Q) allocates a new REC storage block, and sets the POINTER variable Q to point to it. Next, the statement P->NEXT = Q specifies that the NEXT field of the first block (the one pointed to by P) is to point to the same thing that Q points to. The result is two allocated REC blocks, with the first pointing to the second:

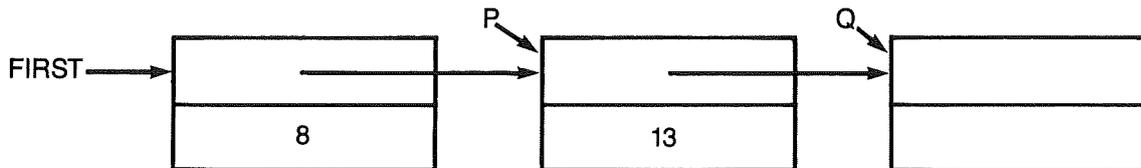


The next statement of the loop inputs a data value, storing it into the NUMB field of the second block, the one pointed to by Q. Then, the assignment statement P = Q advances the pointer variable P so that it

points to the same block pointed to by Q. The result, assuming that the second data value is 13, may be shown as



Subsequent repetitions of the loop allocate additional blocks in the linked list, setting them so that each points to the next one. This process is accomplished by advancing the pointers P and Q as the repetitions continue. For example, halfway through the first repetition of the loop, when K = 3, the linked list may be thought of as follows:



The last line of code in the preceding example illustrates a new built-in function, `NULL()`. This built-in function takes no arguments, and it returns a `POINTER` value that points nowhere. The assignment statement `P->NEXT = NULL()` specifies that the `NEXT` field of the last block in the linked list is to point nowhere.

The following loop prints out all the numbers in the linked list:

```

DO P = FIRST REPEAT(P->NEXT)
    WHILE(P ^= NULL());
    PUT LIST(P->NUMB);
END;
  
```

The `DO` statement specifies that the pointer P begins at the first block in the list, the one pointed to by `FIRST`, and that in each subsequent repetition the pointer P moves up to the next block in the linked list. The repetitions continue until P equals `NULL()`, indicating the end of the linked list.

The complete program for the above discussion follows.

```

LINK: PROCEDURE OPTIONS(MAIN);
      DECLARE 1 REC BASED,
             2 NEXT POINTER,
             2 NUMB FIXED;
      DECLARE (FIRST, P, Q) POINTER;
      ALLOCATE REC SET(FIRST);
      PUT LIST('ENTER FIRST NUMBER');
      GET LIST(FIRST->NUMB);
      P = FIRST;
      DO K = 2 TO 20;
        ALLOCATE REC SET(Q);
        P->NEXT = Q;
        PUT LIST('ENTER ANOTHER NUMBER');
        GET LIST(Q->NUMB);
        P = Q;
      END;
      P->NEXT = NULL();
      PUT SKIP LIST('END OF RUN');
      PUT SKIP;
      DO P = FIRST REPEAT P->NEXT
        WHILE(P ^= NULL);
        PUT LIST(P->NUMB);
      END;
      END LINK;

```

AREA and OFFSET Variables

If you use the `ALLOCATE` statement to allocate several blocks of storage, you have no control over where those blocks of storage are located. In fact, it is possible for PL/I to allocate those blocks of storage in such a way that they are scattered throughout memory. This is not a problem in most applications, because you can use the `POINTER` values to find those storage blocks, no matter where they are. However, suppose you wish to allocate the blocks in such a way that they are all collected in one place, so that you can use an output operation to write all the blocks together out onto secondary storage. For this, you must be able to tell PL/I to allocate all of its `BASED` storage within a single larger region of storage, which we call an area.

PL/I provides this kind of support by means of two noncomputational data types, `AREA` and `OFFSET`. The value of an `AREA` variable is not a value in the usual sense. Rather, the value of an `AREA` variable is a block of storage from which you may suballocate smaller blocks of storage for `BASED` variables. An `OFFSET` variable is like a `POINTER` variable in that you use it with a `BASED` variable to specify the location of the storage for the `BASED` variable. However, it differs from a `POINTER` variable in that it is used only in conjunction with an `AREA` variable, and its value is not the location of the storage block

being allocated, but is rather the displacement or offset of the storage block from the beginning of the area from which it was suballocated.

To understand the relationships among BASED variables, AREA variables, and OFFSET and POINTER variables, look at an example starting with some basic declarations:

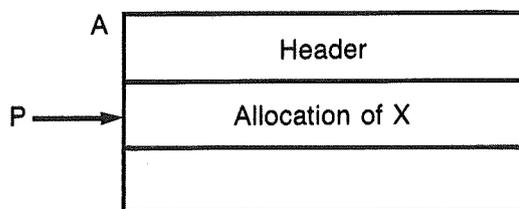
```
DECLARE (A,A2) AREA(2000);
DECLARE O OFFSET;
DECLARE P POINTER;
DECLARE X FIXED BASED;
```

The first declaration specifies that two variables, A and A2, are to be AREA variables, each representing a region of storage containing 2000 bytes. The second line specifies that O is an OFFSET variable. O here represents displacements inside the two AREA variables.

Now consider the following statement:

```
ALLOCATE X IN(A) SET(P);
```

This statement specifies that an allocation of the BASED variable X is to be made within the area A. The result may be pictured as shown in Figure 7-5. In the figure, A represents a region of storage that is one segment or less in size. PL/I uses the first portion of that region as a header in which to store control information about what has been allocated in the area. After the ALLOCATE statement has been executed, a small portion of A is reserved for that allocation of X, and the pointer P points to that allocation.

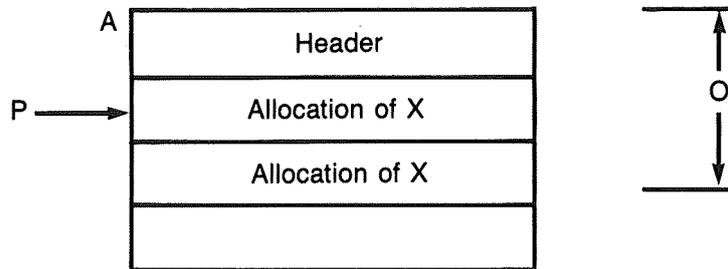


Allocation Within an AREA
Figure 7-5

The statement

```
ALLOCATE X IN(A) SET(O);
```

causes a second allocation of X to be made, but this time the OFFSET variable O is to be set rather than a POINTER variable. The result is shown in Figure 7-6. There are now two allocations of X, and, as indicated by the figure, the value of O is the displacement of the second allocation from the beginning of the area.



Two Allocations of X
Figure 7-6

As with ordinary BASED variables, use the \rightarrow operator to reference a specific allocation of X. For example, the statement

```
P->X = 5;
```

stores the value 5 into the first allocation of X.

On the other hand, a statement like

```
O->X = 5;
```

is not valid, because PL/I does not have enough information. This statement tells PL/I that you want to reference an allocation at a certain displacement from the beginning of an area, but PL/I has no idea which area you mean. The \rightarrow operator must be preceded by a POINTER value. Therefore, in order to reference the second allocation of X, it is necessary to convert the OFFSET variable O to a value. The way to do this is by means of the built-in function POINTER. This function takes two arguments, an OFFSET value and an AREA value, and returns the corresponding POINTER value.

For example, the statement

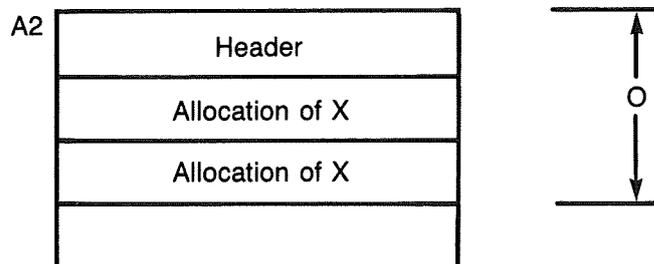
```
POINTER(O,A)->X = 5;
```

is legal, because now you are telling PL/I both the AREA variable, A, and the displacement or OFFSET value within that area.

It is legal to execute an assignment statement involving AREA variables. For example, the statement

```
A2 = A;
```

assigns the entire area A to the area A2. The result is as shown in Figure 7-7. The area A2 now contains two allocations of X. The OFFSET variable O is a valid displacement to the second allocation of X in the area A2. This is the value of OFFSET variables. When you assign one area to another, the OFFSET values remain the same for the individual suballocations.



The OFFSET Variable
Figure 7-7

On the other hand, there is no way to access the first allocation of X in A2. There is no OFFSET variable indicating its displacement, and the POINTER variable P, used in the area A, has no meaning in the area A2.

In this situation, you could actually access the first allocation of X in A2 by a somewhat roundabout method. A built-in function, called `OFFSET`, can be used to convert a `POINTER` variable to an `OFFSET` value. If you used this built-in function on the `POINTER` value P, you would have an `OFFSET` value to the first allocation of X in A that would also be valid for the first allocation of X in A2. That `OFFSET` value could then be used to access the first allocation in A2 in something like the following method:

```
DECLARE O2 OFFSET;  
O2 = OFFSET(P,A);  
POINTER(O2,A2)-> = 10;
```

This group of statements would assign the value 10 to the first allocation of X in A2.

Use the `FREE` statement with the `IN` option to free a suballocation within an area. For example, the statement

```
FREE P->X IN(A);
```

is valid to release the first allocation of X in the area A.

Use the built-in function `EMPTY` if you wish to free all allocations in an area simultaneously. For example, the statement

```
A2 = EMPTY();
```

assigns a cleared area to the `AREA` variable A2, with the effect that all suballocations within A2 are freed.

As we have already stated, an `OFFSET` variable may not be used by itself to indicate the location of a specific allocation. You must always specify which `AREA` variable contains the allocation, and you do this with the built-in function `POINTER`.

In the declaration of an `OFFSET` variable, you may specify a default `AREA` variable, which PL/I is to use when you use the `OFFSET` variable by itself without specifying an explicit `AREA` variable. For example, if the declaration of O given above were changed to

```
DECLARE O OFFSET(A);
```

you could use a reference like $O \rightarrow X$, and PL/I would assume that you meant the area A. In addition, statements like $P = O$ and $O = P$ are legal, since PL/I can supply the appropriate AREA variable. Note that you could use the OFFSET variable O with any other AREA variable, by using either the POINTER or the OFFSET built-in function. For example, the statement

```
POINTER(O,A2)->X = 5;
```

is legal, with the explicit AREA variable A2 replacing the default AREA variable A.

The ALLOCATE and FREE Statements

The format of the ALLOCATE statement is as follows:

```
ALLOCATE specification {,specification}...;
```

where each specification is of the form

```
variable {IN(area-variable)} {SET(locator)}
```

If there are multiple specifications, multiple allocations are done with a single ALLOCATE statement, provided the total allocation does not exceed one segment. PL/I raises the AREA condition if an allocation exceeds one segment.

The variable must be either CONTROLLED or BASED. If it is CONTROLLED, the IN and SET options are illegal.

If the variable is BASED, you must specify a locator, either a POINTER variable or an OFFSET variable. You may specify the locator either explicitly by means of the SET option, or by default if a locator is specified with the BASED attribute in the declaration of the variable.

If you wish the allocation to take place inside an area, you must specify an AREA variable. You may do this either explicitly by means of the IN option, or by default, if the locator is an OFFSET variable with a default AREA variable.

The FREE statement has the format

```
FREE specification {,specification}...;
```

where each specification is of the form

```
{locator->} variable {IN(area-variable)};
```

Multiple freeings take place if there are multiple specifications in the FREE statement. The variable must be either CONTROLLED or BASED. If it is CONTROLLED, the locator and the IN option may not be specified.

If the variable is BASED, you must specify a locator, either explicitly in the specification or else by default if the declaration for the BASED variable contains a default locator. If the freeing is to take place within an area, you must specify the area either explicitly by means of the IN option, or by default if the locator is an OFFSET variable with a default AREA variable.

TECHNIQUES FOR OVERLAYING STORAGE

The term overlying storage describes the situation that occurs when two variables share the same storage area. Of course, one reason to do this might be simply to save space. However, it is more common for a programmer to use these techniques to access the same block of data in two different ways, either considering it as organized into aggregates in two different ways or treating some of the data as having two different data types.

To illustrate the last sentence, consider the two following declarations:

```
DECLARE A(3) FIXED;  
DECLARE 1 S,  
        2 X FIXED,  
        2 Y FIXED,  
        2 Z FIXED;
```

The variable A is an array, and the variable S is a structure. Each of them is an aggregate containing three FIXED data values. If it were possible for A and S to share the same storage area, large enough to hold the three FIXED data values, then the first data value could be accessed by either a reference to A (1) or a reference to S.X. Similarly, the second data value could be referenced by either A (2) or S.Y, and the third by A (3) or S.Z. This ability to organize the same data area into two different aggregate configurations can be very valuable in certain applications.

In a different example,

```
DECLARE C CHARACTER(100);
DECLARE D(100) CHARACTER(1);
```

each of the variables C and D requires 100 characters of storage. If it were possible for C and D to occupy the same storage, you would be able to reference the same characters either as a single long CHARACTER string, or else as members of a CHARACTER array.

As a final introductory example, consider the following declarations:

```
DECLARE F FLOAT BINARY(23);
DECLARE B BIT(32);
```

On the Prime computer architecture, the FLOAT variable F declared above requires two words of storage, or 32 bits. Therefore, if F and B could share the same two words of storage, your program could use F to refer to those two words as a FLOAT number, and could use B to examine the format of that FLOAT number as a BIT string. This is an example of a situation in which the same storage area can be accessed using two different data types.

The two types of overlaying of storage among variables are machine independent and machine dependent. The first two examples just given would be machine independent, because sharing storage in those situations would have the same results on any implementation of PL/I on any machine. The third example, where a FLOAT number is overlaid by a BIT string, is an example of machine dependent overlaying. The reason is that the BIT configuration of FLOAT numbers differs from machine to machine, and so you might get different results on different machines from overlaying that storage.

Overlaying of BASED Storage

Overlaying BASED storage is done fairly easily, since you need only use the same POINTER variable with different BASED variables. Consider, for example, the following declarations:

```
DECLARE P POINTER;
DECLARE C CHARACTER(100) BASED;
DECLARE D(100) CHARACTER(1) BASED;
```

Since you can use the pointer P with either of the BASED variables C or D, you may reference the same block of storage as either a CHARACTER string of length 100, or an array of 100 single characters. Consider, for example, the following statements:

```
ALLOCATE C SET(P);  
PUT LIST(P->C);  
PUT LIST(P->D);
```

The first PUT statement prints out the region of storage as a single CHARACTER string of length 100. The second PUT statement prints 100 individual characters, following the conventions of PUT LIST. Both statements refer to the same section of storage.

Similarly, you can overlay any two BASED variables, even if they have different data types and different aggregate types. Overlaying storage of different types, however, produces machine dependent results.

The ADDR Built-in Function

The preceding examples demonstrate how it is possible to overlay BASED storage with a BASED variable of a different data type or aggregate type. The ADDR built-in function allows you to overlay any storage, STATIC, AUTOMATIC or CONTROLLED, with a BASED variable.

Consider the following program segment:

```
DECLARE C CHARACTER(100) STATIC;  
DECLARE D(100) CHARACTER(1) BASED;  
DECLARE P POINTER;  
..  
P = ADDR(C);  
PUT LIST(P->D);
```

The ADDR built-in function, which is illustrated in this program segment, takes one argument and returns a POINTER value for the storage for that argument. Therefore, the first assignment statement shown above computes the address of the STATIC variable C and assigns that address as a POINTER value to P. Therefore, the PUT statement prints out the CHARACTER string C as an array of 100 individual characters.

By means of the ADDR built-in function, you can obtain a POINTER value for the storage for any variable of any storage class. Therefore, you can always overlay storage of any storage class with a BASED variable of your choice.

Machine Independent Overlaying Rules

Those programmers who plan to run their PL/I programs only on Prime equipment may feel free to overlay storage of one data type with storage of any other data type. However, for some users it is important that their PL/I programs run on different machines and different implementations of PL/I and get the same results. These programmers must follow certain rules if they are going to use overlaying techniques.

There are two forms of machine independent overlaying: simple overlaying and string overlaying. In simple overlaying, both variables are either scalars with the same data type or are aggregates whose scalar values have the same data types in the same order. For example, consider the following declarations:

```

DECLARE 1 S(3) BASED,
        2 A CHAR(20) VAR,
        2 B FIXED,
        2 C CHAR(20) VAR,
        2 D FIXED;
DECLARE 1 T(6) BASED,
        2 X CHAR(20) VAR,
        2 Y FIXED;

```

The aggregates S and T each contain 12 scalar values, with the data types of these scalars alternating between CHARACTER (20) VARYING and FIXED. As a result, if you overlay S and T, your program is still machine independent.

In string overlaying, both variables are NONVARYING strings, either CHARACTER or BIT. Furthermore, all strings involved must be UNALIGNED. Consider, for example, the following declarations:

```

DECLARE CARD CHAR(80) BASED;
DECLARE 1 IMAGE BASED,
        2 NAME CHAR(20),
        2 CODES(25) CHAR(2),
        2 FILLER CHAR(10);

```

Since CARD is a scalar and IMAGE is a structure containing 27 scalars, CARD and IMAGE are not eligible for simple overlaying. However, notice that both variables contain precisely 80 characters. As a result, they fulfill the requirement for string overlaying, and so overlaying these two variables does not make your program machine dependent.

It is possible to combine simple overlaying and string overlaying, and still remain machine independent. For example, consider the following declarations:

```
DECLARE 1 S BASED,  
        2 A(2) FIXED,  
        2 B BIT(40);  
DECLARE 1 T BASED,  
        2 U FIXED,  
        2 V FIXED,  
        2 W(4) BIT(10);
```

S and T do not qualify either for simple overlaying or for string overlaying. However, the first two data items in both S and T are both FIXED, and thus qualified for simple overlaying, and the last portion of both structures consists of 40 BIT values, qualified for string overlaying. As a result, overlaying the two structures is machine independent.

If you attempt to overlay any two data items with different data types, you will get machine dependent results. This includes the situation where one of the data items is a VARYING string and the other is NONVARYING.

The DEFINED Attribute

In preceding paragraphs, we have discussed how you can use a BASED variable to overlay a storage block of one aggregate type or data type with a variable of a different aggregate type or data type. PL/I also provides a more automatic mechanism for overlaying storage, called the DEFINED attribute.

Consider the following example:

```
DECLARE A(100) FIXED DECIMAL(5);  
DECLARE B(10,10) FIXED DECIMAL(5) DEFINED(A);
```

Both A and B contain 100 FIXED DECIMAL(5) data items, but A is a singly dimensioned array and B is a doubly dimensioned array. This fits the definition of simple overlaying described in the preceding section, and so PL/I permits you to specify DEFINED(A) in the declaration of B to indicate that A and B share the same storage. The variable A is said to be the base variable for the DEFINED attribute for B.

ANS PL/I allows only simple overlaying or string overlaying with the DEFINED attribute. Prime PL/I allows overlaying data of any type using the DEFINED attribute, but such overlaying produces machine dependent effects. (Appendix C explains the underlying representations of PL/I

data types on Prime hardware.) The base variable of a DEFINED attribute may be neither DEFINED nor BASED.

When using string overlay defining, use the POSITION attribute to specify that the DEFINED variable starts at a certain character position within the base variable. Consider the following:

```
DECLARE CARD CHARACTER(80);
DECLARE NAME CHARACTER(20) DEFINED(CARD) POSITION(23);
```

In this example, CARD is a card image of 80 characters. The NAME field in that card image occupies columns 23 through 42. The declaration of NAME just above specifies that NAME is the 20-character field of CARD beginning from the twenty-third character.

PL/I supports a third type of DEFINED attribute called iSUB. The purpose of iSUB defining is to give you a way to define arrays in other than linear or rectangular fashion. Consider the example:

```
DECLARE A(3,5);
DECLARE B(5,3) DEFINED(A(2SUB, 1SUB));
```

A and B are both two-dimensional arrays containing 15 scalar elements. The expression in parentheses following the keyword DEFINED specifies how to translate a subscript list for B from the desired subscript list for A. The symbol 1SUB is to be replaced by the first subscript of A, and the symbol 2SUB is to be replaced by the second subscript. For example, if your program contains a reference to B(M + 1, N), this is equivalent to a reference to A(N, M + 1). Therefore, this use of DEFINED has the effect of reversing the way in which the subscripts vary.

Consider a second example:

```
DECLARE MAT(20,20);
DECLARE DIAG(20) DEFINED(MAT(1SUB,1SUB));
```

Here, MAT is a two-dimensional matrix, and DIAG is a one-dimensional array whose elements form a diagonal of the matrix. The expression following the DEFINED keyword specifies that a reference to DIAG(K) is to be replaced by a reference to MAT(K, K).

EXTENT EXPRESSIONS AND INITIAL ATTRIBUTE

This section deals with certain expressions that can appear in DECLARE statements. Two types of such expressions are extent expressions, which are used as dimension bounds for arrays, maximum string lengths, and AREA sizes, and INITIAL expressions, which appear as iteration factors or initialization values in the INITIAL attribute.

This section discusses what happens when there are variables in these expressions.

The INITIAL Attribute

Use the INITIAL attribute in the declaration of a variable in order to specify what initial value the variable should have. PL/I performs the initialization just after it allocates storage for the variable.

For an ordinary scalar variable, specify the initial value in parentheses following the keyword INITIAL, as in the following examples:

```
DECLARE RATE FIXED INITIAL(0);  
DECLARE STR CHAR(20) VAR INIT('XYZ');
```

In the first declaration, the FIXED variable RATE is to be initialized to 0, and in the second declaration the CHARACTER variable STR is to be initialized to 'XYZ'.

If the variable being initialized is an array, you must specify one initial value for each element in the array. For example, the statement

```
DECLARE M(5) BIN FIXED INIT(8,4,2,7,3);
```

specifies that the array M has five elements, which are to be initialized to 8, 4, 2, 7, and 3, respectively. If the array is large, and if you wish to initialize many of the elements of the same value, then you can use a repetition factor. For example, the statement

```
DECLARE VECTOR(100) INIT((100)0);
```

specifies that the 100 elements in the array VECTOR are all to be initialized to 0. A more complex example is the following:

```
DECLARE S(-10:10) INIT((10)-1,0,(10)1);
```

The vector `S` contains 21 elements, with array subscripts varying from -10 to 10. The `INITIAL` attribute specifies that the first 10 of these values are to be initialized to -1, the 11th to zero, and the last 10 to 1.

If the `INITIAL` attribute does not specify enough values, PL/I does not initialize the entire array. For example, the statement

```
DECLARE VECTOR(100) INITIAL((50) 0);
```

specifies that the first 50 elements of the array `VECTOR` are to be initialized to 0, and the last 50 elements are to be left uninitialized. If the `INITIAL` attribute specifies more values than there are elements in the array, PL/I ignores the extra values.

If you specify an asterisk (*) as an initial value, you are indicating to PL/I that you do not wish any initialization for that particular array element. For example,

```
DECLARE VECTOR(100) INIT((5) *, (10) 0, *, (9) 2);
```

specifies that the first five elements of the array `VECTOR` are to be uninitialized. Elements 6 through 15 are to be initialized to 0. Element 16 is to be uninitialized and elements 17 through 25 are to be initialized to 2.

When you are initializing a string array, a special problem arises when you use a repetition factor with a string constant. The problem is that PL/I could confuse the repetition factor for the initialization with a replication factor for the `CHARACTER` string. Consider, for example, the following declaration:

```
DECLARE CA(5) CHAR(50) VAR INIT((5) 'A');
```

The writer of this statement intended that each of the five elements of `CA` be initialized to 'A'. Instead, PL/I interpreted `(5) 'A'` as a replicated string constant equaling 'AAAAA'. The result is that PL/I initializes `CA(1)` to 'AAAAA', and leaves the rest of the array uninitialized. The easiest way for the programmer to accomplish the multiple initialization is to insert a dummy string replication factor of 1, as follows:

```
DECLARE CA(5) CHAR(50) VAR INIT((5) (1) 'A');
```

In this case, PL/I interprets (1) as a string replication factor, and so interprets (5) as an initialization repetition factor.

When you wish to specify initialization for a structure, specify an INITIAL attribute for each of the scalar elements in the structure. Consider, for example, the following:

```
DECLARE 1 S,  
        2 A INIT(0),  
        2 B INIT(5);
```

Here the structure S is initialized by initializing S.A to 0 and S.B to 5.

If you are initializing an array of structures, your initialization list must provide initialization values for all the elements resulting from the inherited dimension. This is shown in the following:

```
DECLARE 1 S(10),  
        2 A INIT((10)0),  
        2 B INIT((5)0, (5)1);
```

Each of S.A and S.B is an array containing 10 elements, as a result of having inherited the dimension from S. As shown, each INITIAL list contains 10 elements for that reason.

Variables in Extent and INITIAL Expressions

Up until now, all extent and initial expressions have been constant. Under certain circumstances, PL/I permits arbitrary expressions to appear.

If the variable being declared is AUTOMATIC, extent and initial expressions may contain variables, provided that the variables are not AUTOMATIC and are declared in the same block. For example, the statement

```
DECLARE A(N) INITIAL((N)0);
```

is legal, provided that the variable N is not AUTOMATIC and is declared in the same block.

Variables are permitted in declarations for CONTROLLED storage as well. There is no similar restriction on these variables.

For both AUTOMATIC and CONTROLLED, the extent expressions and the initial expressions are evaluated at the time that the variable is allocated. The expressions are not evaluated at any other time. Thus, in the example above, if the value of N should change after A has been

allocated, that change would not affect the dimension size of the array A. Even when PL/I frees the allocation of A, it does not re-evaluate the expression; instead, it uses the value of the dimension size, which it saved when it allocated A.

Variable extent and initial expressions are not permitted for STATIC storage.

Variables and REFER Option for BASED Storage

The handling of variables in initial expressions for BASED storage is the same as for CONTROLLED storage. The expressions in the INITIAL attribute are evaluated only when the BASED storage is allocated by means of an ALLOCATE statement.

However, the rules are somewhat different for variables in extent expressions for BASED storage. If a BASED variable contains a variable extent expression, the extent expression is evaluated not only when the BASED variable is allocated, but also whenever the BASED variable is even referenced. Consider the following program segment:

```

DECLARE A(N) BASED, P POINTER;
N = 10;
ALLOCATE A SET(P);
P->A = 5;
N = 7;
PUT LIST(P->A);

```

At the time the ALLOCATE statement is executed, the value of N is 10, and so PL/I allocates 10 words of storage for A. The next statement assigns the value 5 to each of the 10 elements in the array A just allocated. However, when the PUT statement is executed, only seven array elements are printed. The reason is that the value N has been changed to 7, and when PL/I references A in the PUT statement, it uses the current value of N. Notice that if the value of N had been changed so that it was larger than 10, execution of the program would have produced unpredictable results.

PL/I provides one additional capability for variable sized BASED storage. This feature, the REFER option, allows the value of an extent expression to be part of a structure. Consider the following example:

```

DECLARE 1 STOCKITEM BASED,
      2 NAME CHAR(30),
      2 COUNT BIN FIXED,
      2 CODES(N + 1 REFER(COUNT))FIXED;

```

This declaration describes a `BASED` structure called `STOCKITEM` containing three fields. The third of these fields is an array called `CODES`, where the array bound contains two separate specifications, the expression `N + 1` and the identifier `COUNT`, the latter enclosed in parentheses following the keyword `REFER`. PL/I uses the expression `N + 1` when an `ALLOCATE` statement for `STOCKITEM` is executed. This expression determines the size of the array at allocation time. PL/I also stores the value of that expression into the `COUNT` field of `STOCKITEM`. From that point on, whenever your program references `STOCKITEM`, PL/I uses the value of `COUNT` to determine the size of the `CODES` array. This capability is particularly important in situations involving input/output. If your program writes out an allocation of `STOCKITEM`, and then reads the same allocation back at a later time, your program will still know how large the `CODES` array is, since that information is stored in the structure in the `COUNT` field.

The format of an extent expression using the `REFER` option is as follows:

```
expression REFER(identifier)
```

This format can be used in any `BASED` structure in an extent expression for an array bound, a string length, or an `AREA` size. PL/I evaluates the expression when your program executes an `ALLOCATE` statement for the `BASED` structure. The identifier must be a preceding member of the same `BASED` structure. Whenever your program references the `BASED` structure other than in an `ALLOCATE` statement, PL/I uses the member specified by identifier to determine the size of the extent expression.

INTERNAL AND EXTERNAL SCOPE ATTRIBUTES

The scope of a variable is either `INTERNAL` or `EXTERNAL`. The default is `INTERNAL`.

If a variable has either the `STATIC` or `CONTROLLED` storage class, you may declare it to be `EXTERNAL` in addition. Consider the following example:

```
DECLARE X FIXED DECIMAL(5)
        STATIC EXTERNAL INITIAL(0);
```

If this declaration appears in several different external procedures of your program, any reference to `X` in one of those procedures refers to precisely the same variable as a reference to `X` in a different procedure. Therefore, this method allows you to share data values among external procedures without having to pass those values as arguments.

Note that this method does not work unless the declarations of the EXTERNAL variable are identical in all procedures in which it is used. Even the INITIAL attribute must be identical. If the EXTERNAL variable is a structure, all the structure members must be declared with the same attributes in all procedures, although it is permitted that the member names be different in different procedures.

The EXTERNAL attribute may also apply to FILE and ENTRY constants. In fact, EXTERNAL is the default for these constants.

NAMED CONSTANTS AND NONCOMPUTATIONAL VARIABLES

If a value has a noncomputational data type, that value may not be used in ordinary computations such as addition or concatenation. Previous sections have covered the noncomputational data types POINTER, OFFSET, and AREA. The following sections examine the other noncomputational data types in the PL/I language.

Named Constants

You are already familiar with constants in computational data types. For example, 23 is a constant with data type FIXED DECIMAL(2).

For noncomputational data types, there are special rules for how you specify constants. In all cases, the constant is specified by an identifier, just as if it were a variable. The difference is that it is illegal to assign values to these constants. Since these constants are specified by identifiers, they are called named constants.

To declare an identifier to be a LABEL, FORMAT, or ENTRY constant, use that identifier as the statement label for an appropriate statement. If you use an identifier as a label of a FORMAT statement, the identifier is declared as a FORMAT constant. If you use the identifier as the label of a PROCEDURE or ENTRY statement, it is an ENTRY constant. If you use it as a statement label for any kind of statement other than FORMAT, PROCEDURE or ENTRY, it is a LABEL constant.

Use the DECLARE statement to declare either a FILE or ENTRY constant. Here is an example:

```
DECLARE TAPE FILE RECORD INPUT;
DECLARE RANGE EXTERNAL ENTRY;
```

The first of these statements declares TAPE to be a FILE constant, and the second declares RANGE to be an ENTRY constant.

Noncomputational Variables

The following program segment shows how to define variables of these noncomputational data types:

```

    DECLARE L LABEL VARIABLE;
    GET LIST(X);
    IF X = 0 THEN L = ADD;
      ELSE L = PROD;
    GO TO L;
    ...
ADD:...
    ...
PROD:...

```

In this program segment, L is a LABEL variable, meaning that it is a variable to which you may assign LABEL values. The third and fourth lines of this program segment illustrate two such possible assignments, under control of an IF statement. The GO TO statement on the following line transfers control to either ADD or PROD, depending on which of those LABEL values has been assigned to L.

Similarly, you may declare variables of the FORMAT, ENTRY, and FILE data types. Consider the following program segment:

```

    DECLARE (OUTFIL1, OUTFIL2)
      FILE OUTPUT STREAM PRINT;
    OPEN FILE(OUTFIL1), FILE(OUTFIL2);
    DECLARE F FILE VARIABLE;
    GET LIST(X);
    IF X = 0 THEN F = OUTFIL1;
      ELSE F = OUTFIL2;
    PUT FILE(F) PAGE LIST(A + B);

```

In this example, the FILE variable F may be assigned either OUTFIL1 or OUTFIL2. The PUT statement in the last line of the example may perform output to either of these files, depending upon the current value of the FILE variable F.

ADVANCED PROGRAMMING OPTION: POINTER OPTIONS(SHORT)

If your program makes extensive use of POINTER variables, you may wish to economize both on storage and execution time. To do this, you may declare POINTER variables with the SHORT option, using the following syntax:

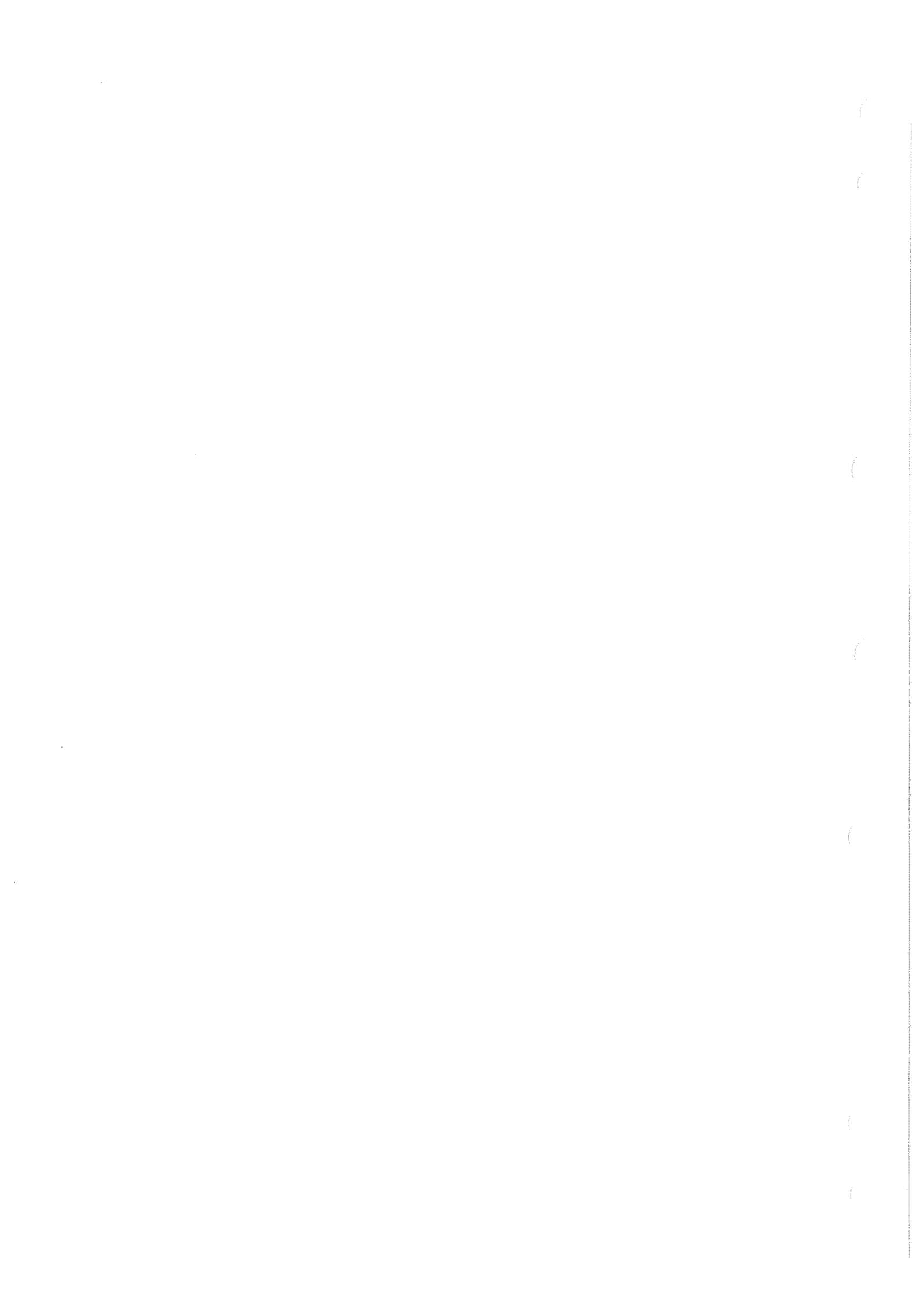
```

    DECLARE P POINTER OPTIONS(SHORT);

```

The pointer occupies two words of storage, rather than three; see Appendix C for more details. The compiler generates instructions that assume no bit offset in the pointer value.

Two-word pointers, having no bit or byte address part, must point to objects aligned on word boundaries. Therefore, you must make sure that a SHORT pointer points to aligned data: BIT and CHARACTER data, for instance, should be declared with the ALIGNED attribute. Because there is no way to ensure that the SUBSTR of a CHARACTER string is word aligned, do not permit a SHORT pointer to point to an object of this type.



8

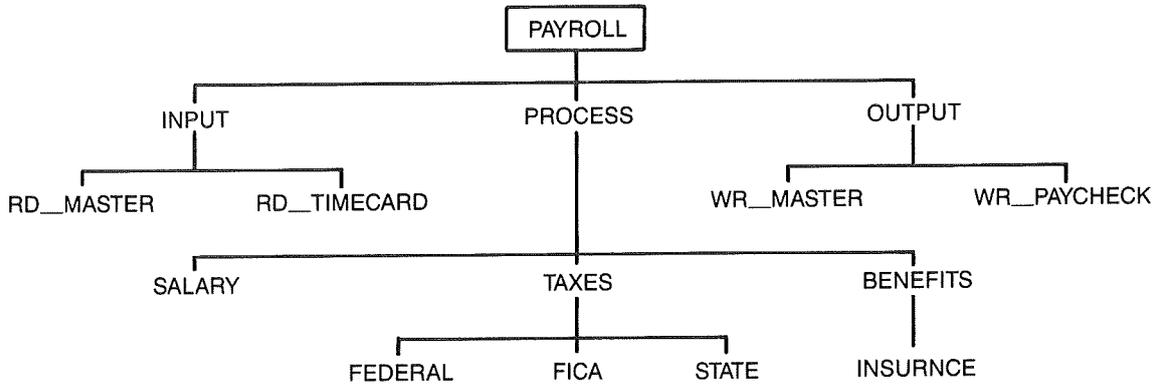
Subroutine and Function Procedures

PROCEDURES

Every PL/I program is a procedure. We have seen numerous examples of the PROCEDURE statement with OPTIONS(MAIN) to indicate the beginning of a main program. If your program is very large, you may find it convenient to break it into smaller programs, each of which performs a single simple task. PL/I gives you this capability by allowing you to break up a large procedure into two or more smaller procedures or subprocedures.

You may be familiar with the terms modular programming and top-down programming, both of which are related to the use of subprocedures. Modular programming refers to the practice of breaking a large program into small pieces, known as modules. This is an important step in the design of a computer program because you can take a large complicated problem and break it up into small components such that each of the components can be programmed in a single small module. It is even possible to assign different modules to different programmers and then later to run all the modules together as one big program.

Top-down programming refers to a type of modular programming where the modules are designed and coded in a certain way. To understand top-down programming, see Figure 8-1.



Top-down Design
Figure 8-1

Figure 8-1 is a hierarchy chart for a large payroll processing program. Each of the boxes on this chart represents a single module in the program. The lines in the chart indicate which modules are called from which modules. For example, PAYROLL calls three modules, INPUT, PROCESS, and OUTPUT. PROCESS itself calls three modules, SALARY, TAXES, and BENEFITS. BENEFITS calls one other module, INSURANCE. Top-down programming of this program has you write the procedures in this chart starting from the top and working down. This means that you write the procedure PAYROLL first, then the modules INPUT, PROCESS, and OUTPUT, and so forth. With this method, you design your program in the most logical style possible. If you code the module PAYROLL first, you are forced to understand the overall flow of control in the program, since this is the highest level module. In general, you work on the conceptually broader modules before you get into the detailed programming required by the lower level modules.

Another purpose of the procedure facility is to save space. Suppose that your program contains the same group of statements in several different places. You may take that group of statements and put them together into a single procedure, and then invoke that procedure from each of the places where the statements have been. This means that the group of statements, which had appeared in several places in your program, now appears only once, in a separate procedure, thereby saving space. The example of a subroutine procedure in the next section illustrates this.

SUBROUTINE PROCEDURES

PL/I permits you to write a procedure as either a subroutine procedure or a function procedure. The difference between these two types is the way in which you invoke them. You invoke a subroutine procedure by means of a CALL statement, and you invoke a function procedure by simply referencing the procedure name in any expression. Function procedures can be very convenient, since they allow you to define your own functions and use them just as you would use the built-in functions that are supplied by the system.

Example of a Subroutine Procedure

Let us start with a simple example of a program that does not use a procedure, and then change it to an equivalent program that does use a procedure. The following program contains no subprocedure, but contains the same group of five statements in two different places. This program uses the GET LIST statement to input an entire array A. The program then prints out the entire array A. Next, the program makes computations that change the array A, and then prints out the array once more. The program then prints END OF PROGRAM and terminates.

```
P:      PROCEDURE OPTIONS(MAIN);
        DECLARE A(100) FIXED DECIMAL(5,2);
        GET LIST(A);

        [ PUT PAGE LIST('PRINTOUT OF ARRAY A');
          PUT SKIP(3);
            DO K = 1 TO 100;
              PUT SKIP LIST(K, A(K));
            END;
          .
          . Computations that change the array A
          .
          .

        [ PUT PAGE LIST('PRINTOUT OF ARRAY A');
          PUT SKIP(3);
            DO K = 1 TO 100;
              PUT SKIP LIST(K, A(K));
            END;

        PUT SKIP LIST('END OF PROGRAM');
        END P;
```

The brackets highlight the fact that the five statements that print out the array A appear in two different places in the program. It is possible to take these five statements, to put them into a separate

subprocedure, and then to use the CALL statement to invoke that subprocedure from each of the two places where the array A is to be printed out.

The result is shown in Figure 8-2. The five statements that print out the array A have been incorporated into a subprocedure called PRNTA, which lies inside the procedure P. There is a CALL statement in each of the two positions where, in the preceding example, the five statements that print out the array A appeared. Each of these CALL statements causes the procedure PRNTA to be invoked, with the result that the five statements that print out the array A are executed.

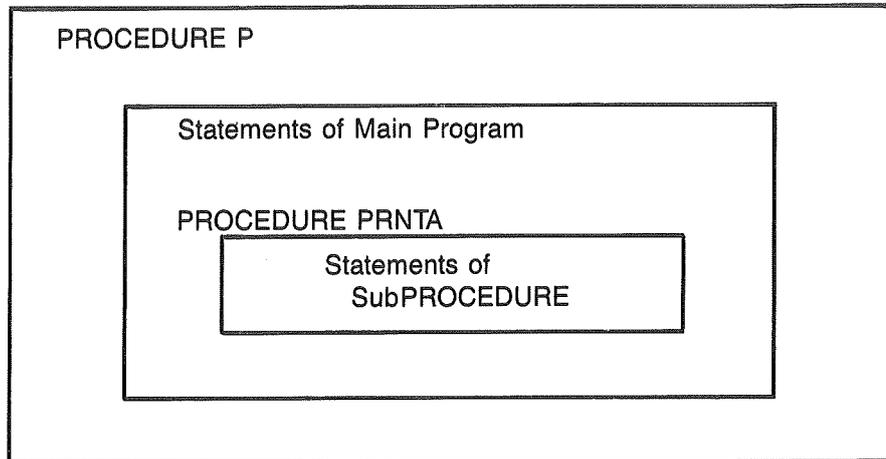
```

P:      PROCEDURE OPTIONS(MAIN);
        DECLARE A(100) FIXED DECIMAL(5,2);
        GET LIST(A);
        CALL PRNTA;
        .
        .
        . Computations that change the array A
        .
        .
        CALL PRNTA;
        PUT SKIP LIST('END OF PROGRAM');

PRNTA:  PROCEDURE;
        DECLARE K BINARY FIXED;
        PUT PAGE LIST('PRINTOUT OF ARRAY A');
        PUT SKIP(3);
        DO K = 1 TO 100;
            PUT SKIP LIST(K, A(K));
        END;
        RETURN;
        END PRNTA;
        END P;
    
```

An Internal Procedure
Figure 8-2

PRNTA is an example of an internal procedure, because it lies entirely inside another procedure. This concept is illustrated by Figure 8-3. As that figure shows, the subprocedure PRNTA is internal to procedure P, since the first lies entirely within the second.

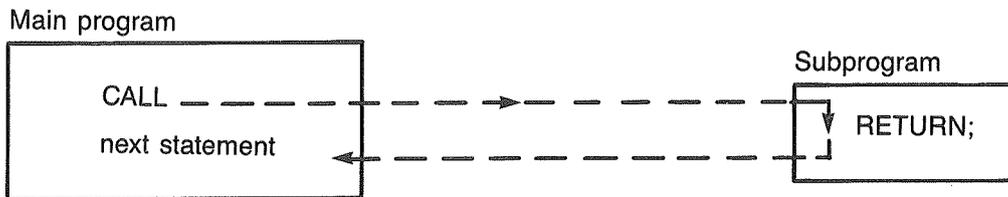


Representation of an Internal Procedure
Figure 8-3

Figure 8-3 also illustrates the distinction between the statements of the main program and those of the subprocedure. The principal flow of control in the program is dictated by the statements in the main program, those statements that are part of procedure P, but are not part of procedure PRNTA. It is only when the main program executes a CALL statement that the statements of PRNTA are even executed.

CALL, PROCEDURE, and RETURN Statements

Figure 8-4 shows what happens when a program executes a CALL statement for a subroutine procedure. This figure pictures the main program and the subroutine as two separate programs, although, in the example above, the subprocedure is physically a part of the main procedure.

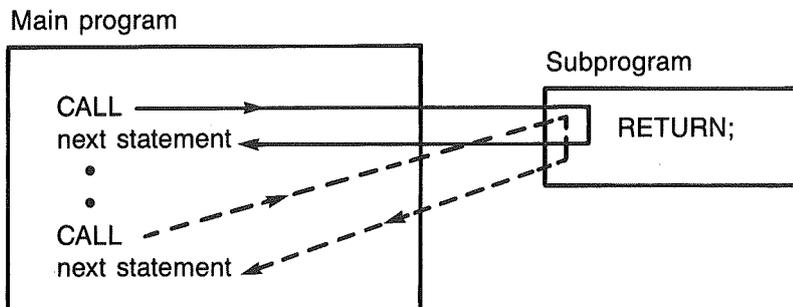


A Call to a Procedure
Figure 8-4

As the figure illustrates, PL/I executes the statements in the main program until it reaches the CALL statement. At that point, PL/I suspends execution of the main program and begins executing the statements of the subprogram. When PL/I reaches the RETURN statement of the subprogram, it terminates execution of the subprogram and resumes execution of the main program. Execution continues with the statement following the CALL statement.

As part of the process that PL/I performs when it executes a CALL statement, PL/I must remember where the CALL statement is located, so that when the RETURN statement in the subprogram is executed, PL/I can return to the correct place in the main program. A more complete picture is shown in Figure 8-5. Execution begins in the main program, and continues until the first CALL statement is reached. At that time, following the line of dashes in the figure, PL/I remembers where the CALL statement is, suspends execution of the main program, and executes the statements in the subprogram. When control reaches the RETURN statement in the subprogram, PL/I returns to the correct point in the main program, according to what it remembers about the location of the CALL statement. Execution of the main program then continues until the second CALL statement is reached. Following the line of dots in the figure, PL/I then suspends execution of the main program again, transferring control to the subprogram. When PL/I reaches the RETURN statement in the subprogram, control returns to the statement following the second CALL statement.

For a subroutine procedure, the CALL statement is the point of invocation of the subroutine procedure. This terminology is used because the subroutine procedure is said to be invoked by the CALL statement.



Multiple Procedure Calls
Figure 8-5

In summary, then, when the main program executes a CALL statement, PL/I performs the following steps:

1. PL/I suspends what it was doing in the main program and remembers where the point of invocation (the CALL statement) is.
2. PL/I invokes the subprocedure and starts to execute the statements within that procedure.
3. When PL/I reaches the RETURN statement in the subprogram, it terminates the subprogram and returns to the point of invocation. This means that PL/I returns to the main program and continues executing statements starting from the statement following the CALL statement.

Dropping Into a Subprocedure

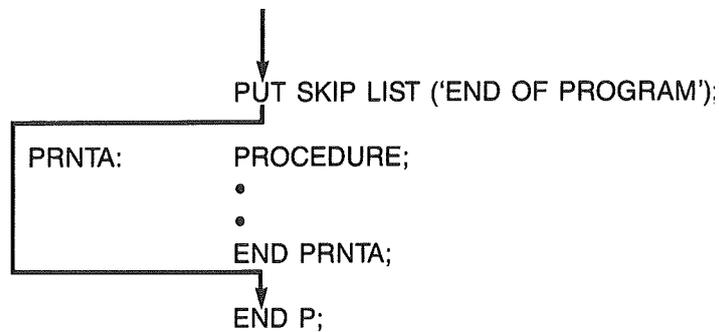
Take another look at Figure 8-2. Notice that the last statement of the main program is

```
PUT SKIP LIST('END OF PROGRAM');
```

This statement immediately precedes the PROCEDURE statement for procedure PRNTA. If your program drops into a subprocedure in this way, PL/I simply skips over the subprocedure. This is illustrated by Figure 8-6. As this figure shows, after PL/I executes the PUT statement, control arrives at the PROCEDURE statement. PL/I simply skips over the entire procedure, and continues execution with the

statement following the END statement for the procedure. In this example, that statement is the END statement for the entire program, so the program terminates at that point.

The general rule, then, is that a program executes the statements of a subprocedure only if that procedure is correctly invoked. This means that a CALL statement is necessary for subroutine procedures, and that an appropriate function reference is necessary for function procedures. (Function procedures are covered later in this chapter.) If your program simply drops into a procedure, PL/I does not execute the statements in the procedure, but rather skips over them.



An Embedded Procedure
Figure 8-6

Notice that this rule implies that you may place your internal procedures anywhere that you wish within your main procedure. Whenever PL/I encounters a procedure within your program, it skips over it. Although PL/I gives you this freedom, good structured programming practice dictates that all of your internal procedures should appear at the end of your main program, just before the final END statement.

Introduction to Scope Rules

Chapter 9 contains a detailed discussion of scope rules, as applied to procedures and other blocks. This section covers enough of these rules to clarify the preceding example and the other examples in this chapter.

In Figure 8-2, which we have already examined, there is a declaration of K within the internal procedure PRNTA. The scope of this declaration is that part of the program to which this declaration applies. If a declaration is made within an internal procedure, it applies only within that internal procedure. In this case, the

declaration of K applies only within procedure PRNTA. If there were a variable called K used in the main program, it would be a different K, just as if the variable name were different. If K were declared in the main program as well, it could have a different data type or aggregate type, without conflicting with the declaration of K within the internal procedure.

On the other hand, consider the array variable A as used within the internal procedure PRNTA. Since there is no declaration of A within the internal procedure, the internal procedure inherits that variable from the main program. That is, when you use A within the internal procedure, it is the same variable A as used within the main program.

The general rule, then, is as follows: if a declaration appears within the main program, its scope is the entire program, excluding those internal procedures in which another declaration of the same variable appears. If a declaration appears within an internal procedure, its scope is only that internal procedure.

Example of a Subroutine Procedure With Parameters

In Figure 8-2, the procedure PRNTA was a subroutine that printed out the entire array A. In the next example, the internal procedure prints out only a portion of the array A. The CALL statement specifies exactly the portion of the array to be printed. This example differs from Figure 8-2 in its CALL statement, in the PROCEDURE statement for the internal procedure, and in the DO statement inside the internal procedure.

```
P:      PROCEDURE OPTIONS(MAIN);
        DECLARE A(100) FIXED DECIMAL(5,2);
        GET LIST(A);
        CALL PRNTA(15, 38);
        .
        .
        . Computations that change the array A and that
        . compute K and L
        .
        CALL PRNTA(K, L);
        PUT SKIP LIST('END OF PROGRAM');

PRNTA:  PROCEDURE (M, N);
        DECLARE K BINARY FIXED;
        PUT PAGE LIST('PRINTOUT OF ARRAY A');
        PUT SKIP(3);
        DO K = M TO N;
        PUT SKIP LIST(K, A(K));
        END;
        RETURN;
        END PRNTA;

END P;
```

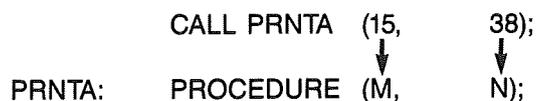
The CALL statement has two arguments, 15 and 38. These arguments represent information that you wish the main program to pass to the internal procedure. Arguments and parameters are defined more fully in the section RELATION BETWEEN ARGUMENTS AND PARAMETERS.

The PROCEDURE statement for the procedure PRNTA has two parameters, M and N. The parameters complete the mechanism for passing information between the main program and the subprogram.

PL/I requires that the number of arguments in the CALL statement must be equal to the number of parameters in the PROCEDURE statement. When PL/I executes the CALL statement, it matches up the arguments and the parameters as illustrated in Figure 8-7. While the statements of the internal procedure are executing, M has the value 15 and N has the value 38. This means that the statement

```
DO K = M TO N;
```

specifies that the loop is to be executed with K going from 15 to 38.



Passing Arguments
Figure 8-7

The parameters M and N have the values 15 and 38, respectively, only while the procedure call is still active. As soon as your program executes the RETURN statement in the internal procedure, these values of M and N are lost. In fact, the scope rules for parameters are just the same as for variables declared within an internal procedure. If there were variables M and N in the main program, they would be entirely different variables, just as if they had different names. These rules are explained in more detail in Chapter 9.

Return now to the programming example, and look at the second CALL statement. By the time control has reached this point in the main program, the values of the array A have presumably been changed, and that values of K and L have been set. PL/I matches these arguments, K and L, with the parameters M and N, respectively, just as in the case of the first CALL statement. During this execution of the internal procedure, the parameters M and N have whatever values your program has computed for the variables K and L. That is, the value of K is assigned to M and the value of L is assigned to N for the duration of this invocation of procedure PRNTA.

SUBROUTINE AND FUNCTION PROCEDURES

The word "assigned" is used loosely in the previous sentence: an assignment is not made in the way that a value is assigned to a regular variable. There is a more complex relationship between an argument and a parameter, which is discussed later in this chapter.

FUNCTION PROCEDURES

PL/I provides a large number of built-in functions, such as ABS and LOG. You may use these functions in any expression to simplify computations.

By means of function procedures, you may also define your own functions to make computations of your choice. You may use these user-defined functions in any expression, as you do built-in functions.

Example of a Function Procedure

The following procedure defines a function called HYP that takes two arguments, representing the sides of a right triangle, and that returns the value of the hypotenuse of that triangle. This function is part of Figure 8-8.

```
P:      PROCEDURE OPTIONS(MAIN);
        ...
        C = HYP(A, B);
        ...
        PUT LIST(HYP(Q + 3, R) + 15);
        ...

HYP:    PROCEDURE(X, Y) RETURNS(FLOAT);
        DECLARE(X, Y) FLOAT;
        DECLARE Z FLOAT;
        Z = SQRT(X * X + Y * Y);
        RETURN(Z);
        END HYP;

        END P;
```

Example of a Function
Figure 8-8

This program contains an internal procedure called HYP. The PROCEDURE statement is slightly different in format from the PROCEDURE statement for a subroutine procedure. It contains two parameters, X and Y, and these are specified the same way as for the subroutine procedure. The

difference is the option RETURNS(FLOAT). This option indicates that HYP is to be invoked as a function rather than as a subroutine. In simplest terms, the implication of this is as follows:

- To invoke the procedure HYP, do not use a CALL statement. Instead, reference HYP just as you would reference an ordinary built-in function. The program example illustrates this with the statement C = HYP(A, B).
- PL/I does not simply return from HYP as it does from a subroutine procedure. Instead it returns a value, the computed value of the function. The option RETURNS(FLOAT) in the PROCEDURE statement for the internal procedure says that HYP returns a float value. In other words, you are defining a function called HYP whose value is FLOAT. The statement RETURN(Z) says that the value of Z computed inside the internal procedure is to be the returned value of the function HYP.

Unlike a subroutine procedure, a function procedure must always take an argument list and contain a parameter list, even if both are empty. If the procedure HYP did not take arguments (for instance, if it obtained its data from user input during execution), it would be referenced by the statement

```
C = HYP();
```

and would begin with the statement

```
HYP: PROCEDURE() RETURNS(FLOAT);
```

In the program example, the main program invokes HYP twice. The first time is with the assignment statement

```
C = HYP(A, B);
```

Notice that HYP is invoked just as if it were a built-in function. When the internal procedure computes that value, PL/I assigns that value to the variable C, as specified by the assignment statement.

The second invocation of HYP in the program example is the following PUT statement:

```
PUT LIST(HYP(Q + 3, R) + 15);
```

This example uses the function HYP in a more complicated setting. While in the preceding example, the arguments of HYP were simple

variables A and B, here the first argument is an expression $Q + 3$. Furthermore, the value returned by HYP is not simply assigned to a variable, but is part of an expression that is to be computed by adding 15 to the value returned by HYP.

These examples illustrate the important fact that, like built-in functions, user-defined functions can interact with expressions in any way you desire. This is true for the following two reasons:

- The arguments to a user-defined function may be arbitrary expressions.
- The user-defined function may be used within an arbitrary expression, and the value returned by the function is used in the computation of that expression.

Arguments and Parameters for Function Procedures

The rules for arguments and parameters for function procedures are identical to the rules for subroutine procedures. The detailed relationships between arguments and parameters are discussed later in this chapter.

The following points clarify the previous example:

- The PROCEDURE statement for HYP specifies two parameters, X and Y. For this reason, any reference to HYP as a function must specify exactly two arguments. PL/I matches up these arguments to the corresponding parameters exactly as in the case of subroutine procedures.
- Notice that the internal procedure contains the following statement:

```
DECLARE(X,Y) FLOAT;
```

The purpose of this DECLARE statement is to specify the data types of the parameters. In the absence of such a declaration, the parameters receive the default data types, BINARY FIXED.

- The arguments can be any expression.

Return Mechanism for Function Procedures

Although subroutine and function procedures are similar in many ways, the difference between the two types is in the way you invoke the procedure and in the way that the procedure returns to the point of invocation. You invoke a subroutine procedure by means of a CALL

statement, and you invoke a function procedure by referencing the procedure name in any expression. In both cases, you specify precisely as many arguments as there are parameters in the PROCEDURE statement. When the procedure ends, a subroutine procedure simply returns to the point of invocation, so that execution can continue with the next statement. On the other hand, a function procedure returns to the point of invocation with a value, the computed value of the function. PL/I normally continues execution by finishing the evaluation of the expression in which the reference to the function procedure occurred. In fact, a single expression may contain several references to function procedures.

Because of the more complicated return mechanism from a function procedure, when you write a function procedure you must specify the data type and value to be returned when the function is invoked.

Specify the data type to be returned by the function procedure with the option

```
RETURNS(descriptor)
```

in the PROCEDURE statement.

The descriptor is the data type of the value to be returned by the function. (The next section explains that the descriptor also specifies the aggregate type to be returned by the function.)

Specify the computed value to be returned by the function procedure by means of the RETURN statement, in a different format from that used with the subroutine procedure. For a function procedure, the RETURN statement has the format

```
RETURN(expression);
```

where the expression is an arbitrary expression. PL/I executes this form of the RETURN statement by evaluating the expression, converting it to the data type and aggregate type specified by the RETURNS option of the PROCEDURE statement, and returning with that value to the point of invocation.

To illustrate the use of an arbitrary expression in the RETURN statement, the following example rewrites only the internal procedure portion of Figure 8-8. This new version of the internal procedure HYP eliminates the need for the auxiliary variable Z by simply computing the expression to be returned entirely within the RETURN statement.

```
HYP: PROCEDURE(X,Y) RETURNS(FLOAT);
      DECLARE(X,Y) FLOAT;
      RETURN(SQRT(X * X + Y * Y));
      END HYP;
```

A user-defined procedure may return any data type that you desire. Specify the desired data type in the descriptor of the RETURNS option of the PROCEDURE statement.

For example, Figure 8-9 is a function procedure that returns a CHARACTER(1) value. A reference to LET(K) returns the Kth letter of the alphabet. The statements

```
K = 3;
PUT LIST(LET(K));
```

print the third letter of the alphabet, C. Note that there is a little "defensive programming" in the fourth line. If the argument is not positive, or if it is greater than 26, then LET returns a dash.

```
LET: PROCEDURE(N) RETURNS(CHAR(1));
      DECLARE ALPH CHARACTER(26)
            STATIC INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ');
      IF N <= 0 | N > 26
      THEN RETURN('-');
      ELSE RETURN(SUBSTR(ALPH, N, 1));
      END LET;
```

A Function That Returns a CHARACTER Value
Figure 8-9

As explained in Chapter 6, when PL/I evaluates an expression it allocates targets in which to store intermediate results. PL/I uses the same techniques for references to function procedures. The following steps explain in detail how the return mechanism from a function procedure works.

1. When the user-defined function is referenced in an expression, PL/I allocates a temporary storage location for storage of the value of the function. The size of this storage location is determined at compile time from the data type information specified in the descriptor for the RETURNS option of the PROCEDURE statement.

2. PL/I invokes the function, passing to it information about the location of the newly allocated temporary storage location. PL/I then starts executing statements inside the function.
3. When program execution reaches the RETURN statement of the function, PL/I evaluates the expression in the RETURN statement, converts the result to the data type and aggregate type specified by the descriptor in the RETURNS option of the statement for the function, and stores the resulting data value in the temporary target storage area described in step 1.
4. PL/I then returns control to the point of invocation, the place where the function was called. At that point, PL/I continues to evaluate the expression in which the function reference occurred. In order to evaluate that expression, PL/I finds the value of the function reference in the temporary target storage area allocated in step 1.

Note

The above steps are changed when there is an asterisk in the RETURNS descriptor. The differences are described in a later section.

Functions That Return an Array or Structure

In all the discussion and examples of function procedures so far, the procedure has returned a scalar value. This means that the function has returned, for example, a single FLOAT or a single CHARACTER value.

In fact, you may define functions that return any PL/I aggregate type. Specify the aggregate, just as you specify the data type, in the descriptor field of the RETURNS attribute in the PROCEDURE statement for the function procedure.

The following example illustrates a function procedure that returns an array value. This function, ADD5, has a single parameter that is a FLOAT array with a dimension size of 10. The function returns a FIXED array, also with a dimension size of 10.

```
ADD5: PROCEDURE(ARR) RETURNS((10) FIXED);  
      DECLARE ARR(10) FLOAT;  
      RETURN(ARR + 5);  
      END ADD5;
```

The RETURN statement in the third line of the function procedure specifies that the value to be returned is ARR + 5. This means that when the function procedure terminates and control returns to the point of invocation, the value returned is obtained by adding 5 to each

SUBROUTINE AND FUNCTION PROCEDURES

element of the parameter array and converting each of those elements to FIXED. The result is a FIXED array returned to the point of invocation.

To see how the function ADD5 might be invoked, consider the following statements that might appear in the main program:

```
DECLARE A(10) FLOAT, B(10) FIXED;
...
PUT LIST(ADD5(A));
B = ADD5(A);
```

First look at the PUT statement in this example. The argument is the FLOAT array A, which is passed to the ADD5 procedure, to be matched up with the parameter ARR. The value returned is the result obtained by adding 5 to each element of A and converting the result to FIXED. Therefore, the PUT statement prints out an array of 10 FIXED values.

The assignment statement on the next line references the same function, with the same argument, but does not print it out. Instead, the array value returned by ADD5 is assigned to the array B.

Similarly, it is possible for a function procedure to return a structure aggregate value. Figure 8-10 is such a function procedure. Since this is a more complicated example than any given previously, look at it in some detail.

FIND is a function procedure that has a CHARACTER(20) parameter and that returns a structure value. Examine the RETURNS option in the PROCEDURE statement:

```
RETURNS(1,2 CHAR(20),2 FIXED DEC(7,2),
        2 PICTURE 'A9X99');
```

The descriptor in this RETURNS attribute specifies a structure value containing three individual scalar values. The three individual scalar values have data types CHARACTER(20), FIXED DECIMAL(7,2), and PICTURE 'A9X99'. If this RETURNS descriptor is confusing to you, compare it to the declaration of the structure S in the same example. Notice that the RETURNS descriptor is identical to the declaration of S, except that the descriptor contains no variable names, either the name of the structure or the names of any of the members. This is a useful trick for figuring out how to write a descriptor: just write what you want as an ordinary declaration, and delete the names of the variables.

```

FIND: PROCEDURE(NAME) RETURNS(1,
      2 CHAR(20), 2 FIXED DEC(7,2),
      2 PICTURE 'A9X99');
DECLARE NAME CHARACTER(20);
DECLARE 1 S,
      2 NAME CHAR(20),
      2 PRICE FIXED DEC(7,2),
      2 CODE PIC 'A9X99';
DECLARE K FIXED BINARY;
DECLARE NAMLIST(5) CHAR(20) STATIC
      INIT('TOY', 'BOAT', 'CLOCK', 'BOOK', 'PEN');
DECLARE PRLIST(5) FIXED DEC(7,2) STATIC
      INIT(7.43, 12.52, 8.92, 10.53, 7.50);
DECLARE CDLIST(5) PIC 'A9X99' STATIC
      INIT('C3-42', 'C5-25', 'C4-99',
          'D9-42', 'X3-25');
DO K = 1 TO 5 WHILE(NAME ^= NAMLIST(K));
      S.NAME = NAME;
      END;
IF K <= 5 THEN DO;
      S.PRICE = PRLIST(K);
      S.CODE = CDLIST(K);
      END;
ELSE DO;
      S.PRICE = 0;
      S.CODE = 'Z9-99';
      END;
RETURN(S);
END FIND;

```

A Function That Returns a Structure
Figure 8-10

The FIND function is intended to be a rudimentary inventory control routine. It takes one argument, the name of a product, and it returns a structure containing three values: the name of the product, the price of the product, and the inventory code for the product.

Inside the procedure FIND are declarations of three arrays. NAMLIST is an array that is initialized to a list of all the products supported by this rudimentary procedure. PRLIST and CDLIST are arrays that are initialized to the prices and inventory codes, respectively, of all the products in the NAMLIST array.

The procedure contains a two-line repetitive DO group, whose purpose is to search for the product specified by the parameter NAME in the list provided in the array NAMLIST. At the end of this loop, K equals the index of the product within the array NAMLIST if the search is successful, and K equals 6 if the search is unsuccessful.

The next to last line of the procedure is the statement

```
RETURN(S);
```

This statement returns from the function procedure, passing as a value the current value of the structure S, whose members have been assigned the name, price, and inventory code of the product by the preceding statements in the procedure.

For example, suppose that FIND is invoked by means of the statement

```
PUT LIST(FIND('CLOCK'));
```

When the procedure is invoked, the repetitive DO group computes a value of $K = 3$, since CLOCK is the value of the third element NAMLIST array. S.NAME is set equal to 'CLOCK', S.PRICE is set equal to 8.92, and S.CODE is set equal to 'CA-99'. The value returned by FIND is a structure containing these three scalar values, and the PUT statement that invokes FIND prints out these three scalar values. Similarly, it would be possible to invoke FIND from an assignment statement that assigned the structure returned by FIND to another structure variable.

In summary, the RETURNS descriptor can specify any aggregate type and any data types. Although you usually use only scalar RETURNS descriptors, you may occasionally define a function that returns an array, a structure, or an array of structures.

RETURNS Descriptor With Variable Extent Expressions

As defined in Chapter 7, the term extent expression refers to an expression, usually a constant, that you specify in a DECLARE statement for a string length, an array bound, or an AREA size. For example, in the DECLARE statement

```
DECLARE CA(5) CHARACTER(20);
```

there are two extent expressions, the array bound 5 and the string length 20. In this example, both extent expressions are constant, which is the usual case.

Extent expressions also appear in RETURNS descriptors to represent string lengths, array bounds, and AREA sizes. In all the examples given previously, these extent expressions have been constant.

The restriction of extent expressions in RETURNS descriptors to constants can be very inconvenient, particularly when the value being

returned is a CHARACTER string. The problem is that when you are writing the function procedure, you do not know what the maximum string size will be when the function procedure is executing with various arguments. In such cases, PL/I permits you to use an asterisk for the extent expression in the RETURNS descriptor to indicate that the extent expression is not known at the time the program compiles, but will be determined each time the function procedure is referenced, depending upon the value returned by the RETURN statement of the function.

This point can be illustrated with a modification of a previous example. Earlier, this section examined Figure 8-9, which defined a function procedure that returned a CHARACTER(1) value equal to the letter of the alphabet specified by the numeric argument. For example, LET(3) would return the value 'C', the third letter of the alphabet. This example can be modified to create a new function, LET2, which takes two arguments. The first argument is as before, and the second argument is the number of copies of the letter of the alphabet that you wish returned. For example, a reference to LET2(4,6) would return a CHARACTER string of length 6 containing six occurrences of the fourth letter of the alphabet, or 'DDDDDD'. This is a simple example of a function procedure that returns a CHARACTER string whose length is not known at compile time, since LET2 may be called with any second argument. Therefore, it is impossible to specify a constant string length in the RETURNS descriptor for LET2.

The problem can be handled as shown in the example below. The RETURNS descriptor specifies CHAR(*), indicating that the function can return a CHARACTER value of any length, where the length is determined anew each time the function procedure is invoked.

```
LET2: PROC(N,L) RETURNS(CHAR(*));
      DECLARE C CHARACTER(1);
      DECLARE ALPH CHARACTER(26) STATIC
            INITIAL('ABCDEFGHIJKLMNOPQRSTUVWXYZ');
      IF N <= 0 | N > 26 THEN C = '-';
      ELSE C = SUBSTR(ALPH, N, 1);
      RETURN(COPY(C, L));
      END LET2;
```

The length of the string that is returned is determined by the RETURN statement in the next to last line of the function procedure. As you can see, the expression specified with this RETURN statement yields a string of length L. CHAR(*) is appropriate in the RETURNS descriptor, because you do not know when you write this procedure what the value of L will be when the procedure is called.

Similarly, you can specify an asterisk for any extent expression in the RETURNS descriptor, string length, array bound, or AREA size. Furthermore, if the RETURNS descriptor is for a structure with several members, any of the members can have an asterisk in its individual extent expression.

SUBROUTINE AND FUNCTION PROCEDURES

Notice that the steps outlined above in the section Return Mechanism for Function Procedures do not apply when there is an asterisk in the RETURNS descriptor. The problem is that PL/I cannot allocate the temporary target area for the returned value when the function is invoked, since the size of this target is not known until the RETURN statement is executed. For that reason, PL/I postpones allocation of the temporary storage area for the target until the RETURN statement is executed. PL/I uses special techniques so that when your program returns from the function procedure to the point of invocation, PL/I can continue evaluating the expression in which the function reference appeared by using special information on where the temporary target storage area can be found.

SUMMARY OF DIFFERENCES BETWEEN SUBROUTINE AND FUNCTION PROCEDURES

The following list is a summary of the differences in subroutine and function procedures:

- A subroutine procedure is invoked by means of a CALL statement. A function procedure is invoked by means of a reference to the function name in any expression.
- The RETURNS option of the PROCEDURE statement is forbidden for a subroutine procedure; it is required for a function procedure.
- Arguments and parameters are handled the same for subroutine and function procedures.
- The RETURN statement may not include an expression for a subroutine procedure; it must include an expression for a function procedure.
- Executing the END statement of a procedure is equivalent to a RETURN statement for a subroutine procedure; it is illegal for a function procedure.

RELATION BETWEEN ARGUMENTS AND PARAMETERS

The preceding pages of this chapter gave a number of examples of subroutine and function procedures using arguments and parameters. The following sections examine the detailed rules covering arguments, parameters, and the relationships between them. Bear in mind that all of these rules are identical for both function and subroutine procedures.

In many mathematical, scientific, or engineering applications, the terms argument and parameter mean the same thing. In the PL/I language, these two terms have different meanings. An argument appears in the statement that invokes a procedure, whether the invocation is by means of a CALL statement for a subroutine procedure, or a function

reference for a function procedure. A parameter appears in the PROCEDURE statement. PL/I requires that the number of arguments in the procedure invocation equal the number of parameters in the PROCEDURE statement. When PL/I invokes the procedure, it matches up the arguments in the invocation with the corresponding parameters in the PROCEDURE statement. This section describes how PL/I does this.

How PL/I Handles Parameters

The statement that K has the value 5 means different things depending on whether K is a parameter or an ordinary variable. If K is an ordinary STATIC or AUTOMATIC variable, it means that there is a storage area associated with K, and that the value 5 has been stored in that storage area. If PL/I executes a statement in your program that requires the value of K, PL/I simply fetches the value stored in the storage area associated with K.

It is quite different when the variable is a procedure parameter. PL/I handles parameters quite differently from ordinary variables. This section explores how PL/I handles parameters, and what exactly it means to say that a parameter has such and such a value.

Consider the next example. K is an ordinary variable to which the first statement assigns the value 5. The CALL statement on the second line uses K as an argument. When PL/I invokes the procedure SB, PL/I matches this argument with the parameter M. Since K has the value 5, the parameter M also has the value 5 during execution of the procedure.

```
      K = 5;
      CALL SB(K);
      .
      .
SB:   PROC(M);
      T = 10;
      M = 10;
      PUT LIST(K);
      RETURN;
      END SB;
      .
      .
```

However, the statement that M has the value 5 means something quite different in concept from the statement that K has the value 5. M is not a variable in the ordinary sense, but is a parameter. When PL/I executes the CALL statement, PL/I does not store the value 5 in the storage area associated with M; instead, PL/I sets M as a pointer back to the argument K.

This is the major difference between a parameter and an ordinary variable. A parameter does have a storage area associated with it, but

SUBROUTINE AND FUNCTION PROCEDURES

that storage area is not used to store the value of the parameter. Instead, that storage area is used to store a pointer to the argument. Notice further that the term pointer is used here in an informal sense; it does not refer to a POINTER variable.

Therefore, M does not equal 5 in the usual sense, but really names a pointer to the argument K. Therefore, a reference within the procedure to M is treated as a reference to the argument K.

This concept is illustrated by Figure 8-11 below. This figure shows the storage associated with the variable K and the parameter M. Both K and M can be said to have the value 5, but the meaning of that statement is different for the two cases. The storage associated with K actually contains the value 5, but the storage associated with M contains a pointer to K.

To understand the implications of this concept, consider the statements of procedure SB. First, look at the statement

```
T = 10;
```

This is an ordinary assignment statement, which assigns the value 10 to the variable T. PL/I executes this statement by storing the value 10 in the storage area associated with T.

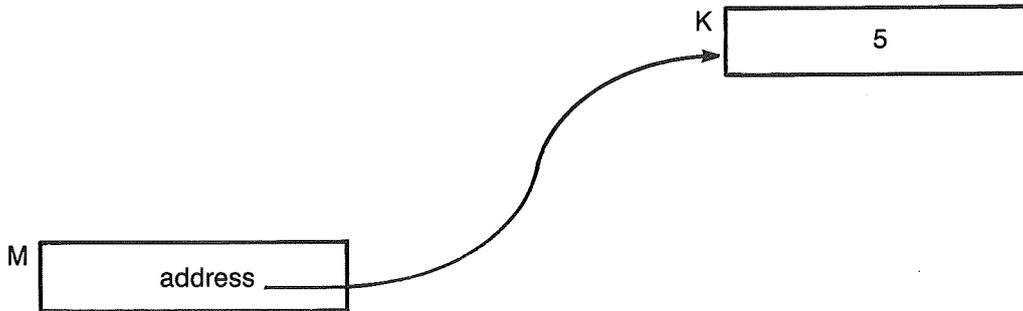
But now look at the statement on the following line:

```
M = 10;
```

This is also an ordinary assignment statement, but now the value 10 is being assigned to a parameter M. PL/I executes this statement not by storing the value 10 in the storage area associated with M, but by storing the value 10 in the storage area pointed to by the area associated with M. As you can see in Figure 8-11, this has the effect of changing the value of K to 10. As a result, the statement on the next line,

```
PUT LIST(K);
```

prints the new value of K, 10.



Storage of Parameters and Values
Figure 8-11

Variable Parameter Extent Expressions

An extent expression in a declaration specifies a string length, array bound, or AREA size. Most of the time, an extent expression is an ordinary constant.

Suppose that you wish to write a subroutine or function procedure that you may invoke with a string argument of any length. When you declare the parameter inside the procedure, if you use a constant for the string length, your procedure is limited to arguments of that string size or maximum string size. Similarly, suppose you wish to write a procedure that you would like to call with an array argument with any upper bound or lower bound. Specifying constants for the array bounds in the declaration of the array parameter would restrict you to arrays of that dimension size.

Under such circumstances, PL/I permits you to use an asterisk for the extent expression in the declaration of the parameter. When you use an asterisk for a string length or array bound, you are telling PL/I that it should assume that the length of the string parameter, or the dimension size of the array parameter, is the same as that for the argument that is matched with that parameter.

This is particularly important in the case of CHARACTER string parameters. You need a procedure that can handle strings of any length in the argument. The next example illustrates this concept.

SUBROUTINE AND FUNCTION PROCEDURES

```
INDEX: PROC(C, S) RETURNS(BIN FIXED);
      DECLARE(C, S) CHAR(*);
      DECLARE K BIN FIXED;
      IF LENGTH(S) > 0 THEN
        DO K = 1 TO LENGTH(C) - LENGTH(S) + 1;
          IF SUBSTR(C, K, LENGTH(S)) = S
            THEN RETURN(K);
        END;
      RETURN(0);
END INDEX;
```

This example shows how you could write the built-in function INDEX as a function procedure. There are two parameters, C and S, both of which are declared with

```
DECLARE(C,S) CHAR(*);
```

This declaration specifies that the lengths of C and S are simply to be taken from the lengths of the arguments that are matched with these parameters. The result is that this INDEX function procedure can be used with arguments of any length. Notice that, in the procedure, the built-in function LENGTH is used to determine the actual length of the arguments matched with the parameters C and S.

You may wish to write a user-defined function that can take an argument string of any length and return a CHARACTER string of any length. In such cases, use CHARACTER(*) in both the parameter data type and the RETURNS descriptor, as in the next example. The UPCASE function is invoked with a CHARACTER string argument of any length; it returns the same string with all the lowercase letters translated to uppercase.

```
UPCASE: PROC(C) RETURNS(CHAR(*));
      DECLARE C CHAR(*);
      DECLARE LOWALF CHAR(26) STATIC
        INIT('abcdefghijklmnopqrstuvwxyz');
      DECLARE UPALF CHAR(26) STATIC
        INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ');
      RETURN(TRANSLATE(C, UPALF, LOWALF));
END UPCASE;
```

The section on array parameters below illustrates the use of the asterisk in the array bounds of an array parameter.

Dummy Arguments

As we have previously explained, PL/I does not handle a parameter the way it handles other variables. The storage associated with a

parameter does not contain the value of the parameter, but rather contains a pointer to the corresponding argument, which was matched to that parameter when the procedure was invoked. (Recall that the term pointer is used in an informal sense here and should not be confused with POINTER variables.) An important result is that when the procedure assigns a new value to the parameter, it is the corresponding argument that is changed.

Sometimes the argument is such that it does not make sense for the parameter to point to it. In such cases, PL/I creates a dummy argument, which the parameter can point to.

There are three cases when PL/I creates a dummy argument:

- When the argument is a constant.
- When the argument is an expression, but not a simple variable reference. This includes the special case of a variable reference that is enclosed in a set of parentheses.
- When the data type or the aggregate type of the argument is different from the data type or aggregate type, respectively, of the parameter.

To understand the concept of the dummy argument, look at the following example. This program contains an internal procedure, S, with a single parameter. The internal procedure contains an assignment statement, L = 25, which assigns a value to the parameter.

```
P: PROC OPTIONS(MAIN);
  DECLARE K FIXED;
  ...
  K = 53;
  CALL S(K);
  ...
  CALL S(53);
  ...
S: PROCEDURE(L);
  DECLARE L FIXED;
  L = 25;
  ...
  RETURN;
END S;
END P;
```

There are two CALL statements that invoke S. In the first one, the argument is K, which has been assigned the value 53. When control passes to the procedure S, the parameter L points to the argument K. As a result, the assignment statement L = 25 changes the value of K to 25.

SUBROUTINE AND FUNCTION PROCEDURES

In the second CALL statement, the argument is a constant 53. It would not make sense for the parameter L to point to a constant. As a result, PL/I allocates a block of storage to be used as a dummy argument. This block of storage is large enough to hold a FIXED value. PL/I stores the value 53 in this dummy argument before invoking S. When PL/I invokes S, L points to the dummy argument that was created. After the RETURN statement is executed, PL/I returns control to the point of invocation, and the storage occupied by the dummy argument is released. As a result, there is no permanent effect of the assignment statement L = 25. In fact, all computations that affect the value of the dummy argument are lost.

Suppose the same program contained the following statements:

```
DECLARE X FLOAT;  
X = 46.2;  
CALL S(X);
```

In this CALL statement the argument X is FLOAT, while the corresponding parameter L is FIXED. Because the data types do not match, it does not make sense for the parameter L to point to the argument X. As a result, PL/I creates a dummy argument. That is, PL/I allocates a FIXED temporary storage area, converts X to the FIXED value of 46, and stores that FIXED value in the storage area. This temporary storage area is the dummy argument to which the parameter L points during execution of the internal procedure S. The statement L = 25 changes the value of the dummy argument, but does not affect the value of X. When the RETURN statement is executed, PL/I frees the storage occupied by the dummy argument and X still equals 46.2.

When an argument is an expression that is not a simple variable reference, PL/I must create a dummy argument. For example, suppose the same program contained the following statement:

```
CALL S(K + 3);
```

It could not make sense for the parameter L to point to the expression K + 3. As a result, PL/I creates a dummy argument, computes the value of K + 3, and stores that value in the dummy argument. During execution of the internal procedure S, L points to this dummy argument, and the value of K is not affected by assignment to the parameter.

A special case of dummy argument creation is the enclosure of a variable in parentheses. An example is

```
CALL S((K));
```

PL/I Reference Guide

Even though K is a variable that has the same data type as the parameter L , PL/I creates a dummy argument, because K is enclosed in an extra set of parentheses. As a result, the statement $L = 25$ does not change the value of K .

On the other hand, the argument may be an element of an array without PL/I's creating a dummy argument. For example, suppose the same main program contained the following two statements:

```
DECLARE KA(100) FIXED;  
CALL S(KA(4));
```

Since $KA(4)$ has the same data type as the parameter L , PL/I creates no dummy argument. The statement $L = 25$ inside the internal procedure changes the value of the array element $KA(4)$.

In fact, the argument subscript can be any expression. For example, the statement

```
CALL S(KA(Z + 3));
```

invokes the procedure S without the creation of any dummy argument. In that case, the parameter L points to whatever array element is indicated by the value of the expression $Z + 3$.

Similarly, the argument may be any element of a structure or an array of structures. If it has the same data type as the parameter, no dummy argument is created.

In all the examples so far in this chapter, the parameter was a scalar. When the parameter is a scalar, the corresponding argument must also be a scalar. If the parameter is a nonscalar aggregate, the argument must be promotable to the aggregate type of the parameter, according to the rules given in Chapter 6 for aggregate promotion. If the aggregate type of the argument does not equal the aggregate type of the parameter, PL/I creates a dummy argument, converts and promotes the argument to the data type and aggregate type of the parameter, and stores the result in the dummy argument. The aggregate parameter then points to the dummy argument.

Array Parameters

When the parameter is an array, the corresponding argument must either be an array or be promotable to an array. If it is not an array, or if it is an array with a data type different from the data type of the parameter, then a dummy argument is created. The rules for the argument or dummy argument are as follows:

- The number of dimensions of the argument must equal the number of dimensions of the parameter. For example, if the parameter is a three-dimensional array, the argument must also be a three-dimensional array.
- For each dimension, the lower bound and upper bound of the argument must equal the lower bound and upper bound, respectively, of the parameter. Alternatively, the declaration of the parameter may contain an asterisk for the dimensions, in which case the parameter will match any lower bound and upper bound in the argument.

The example below contains an internal procedure with a parameter that is declared to be an array with an asterisk for a dimension size. The main program contains two CALL statements to this internal procedure. For these two calls, the arguments are the arrays Q and R, respectively. Since the parameter has an asterisk for an array bound, no dummy argument is created. The upper and lower bounds of the parameter A depend on the upper and lower bounds for the argument. For the first CALL, when the argument is Q, the lower bound for A is 1 and the higher bound is 10. With the second CALL, when the argument is R, the lower bound is 12 and the upper bound is 28.

```

P:  PROC OPTIONS(MAIN);
    DECLARE Q(10);
    DECLARE R(12:28);
    ...
    CALL PRNTAR(Q);
    CALL PRNTAR(R);
    ...
PRNTAR: PROC(A);
    DECLARE A(*);
    DECLARE K BINARY FIXED;
    PUT PAGE LIST('ARRAY PRINTOUT');
        DO K = LBOUND(A,1) TO HBOUND(A,1);
            PUT SKIP LIST(K,A(K));
        END;
    RETURN;
END PRNTAR;
END P;

```

The procedure PRNTAR in this example uses the built-in functions LBOUND and HBOUND. Use these built-in functions in any circumstances where the bounds of an allocated array are not known at the time you are

writing the program. This can happen for a CONTROL or AUTOMATIC array with variable extent expressions, or it can happen in the current case with a parameter, where the array bound is specified by an asterisk.

There are three related built-in functions for use in such circumstances:

- LBOUND(array,n)
- HBOUND(array,n)
- DIMENSION(array,n) or DIM(array,n)

For each of these built-in functions, the array is an array variable and n is the dimension number for which the information is to be computed. In the example above, the parameter A is a one-dimensional array. Therefore, LBOUND(A,1) refers to the first (and only) dimension of the parameter A.

The information computed by the three built-in functions is as follows: LBOUND returns the lower bound of the specified dimension, HBOUND returns the upper bound, and DIMENSION or DIM returns the value of the dimension size, which equals (upper bound - lower bound + 1).

In the preceding example, consider the statement

```
DO K = LBOUND(A,1) TO HBOUND(A,1);
```

The statement specifies that the value of K is to vary from the lower bound to the upper bound of the array A in the first dimension. The values of the lower bound and the upper bound depend upon the parameters, since the declaration for A contains an asterisk for the dimension size. As a result, for the first CALL statement, when Q is the argument, this DO statement executes with K going from 1 to 10. For the second CALL statement, when R is the argument, the value of K goes from 12 to 28. In either case, the result is that K ranges over all possible values of the subscript for the argument array, with the further result that this DO loop prints out all values in the array.

If the parameter is a one-dimensional array, you may use as an argument a cross section of a multidimensional array. For example, suppose the main program of the last example contained the following statements:

```
DECLARE S(5, 15, 25);  
...  
CALL PRNTAR(S(*, I + 3, 1));
```

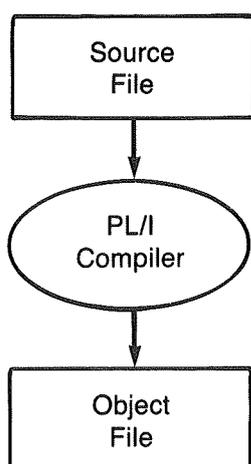
The specified cross section of the three-dimensional array S is passed as a one-dimensional array argument to the procedure PRNTAR. Within PRNTAR, the DO statement is executed with K going from 1 to 5.

You may use appropriate cross sections in any other circumstances as well. For example, you may use a two-dimensional cross section of a four-dimensional array as an argument, when the parameter is a two-dimensional array.

EXTERNAL PROCEDURES

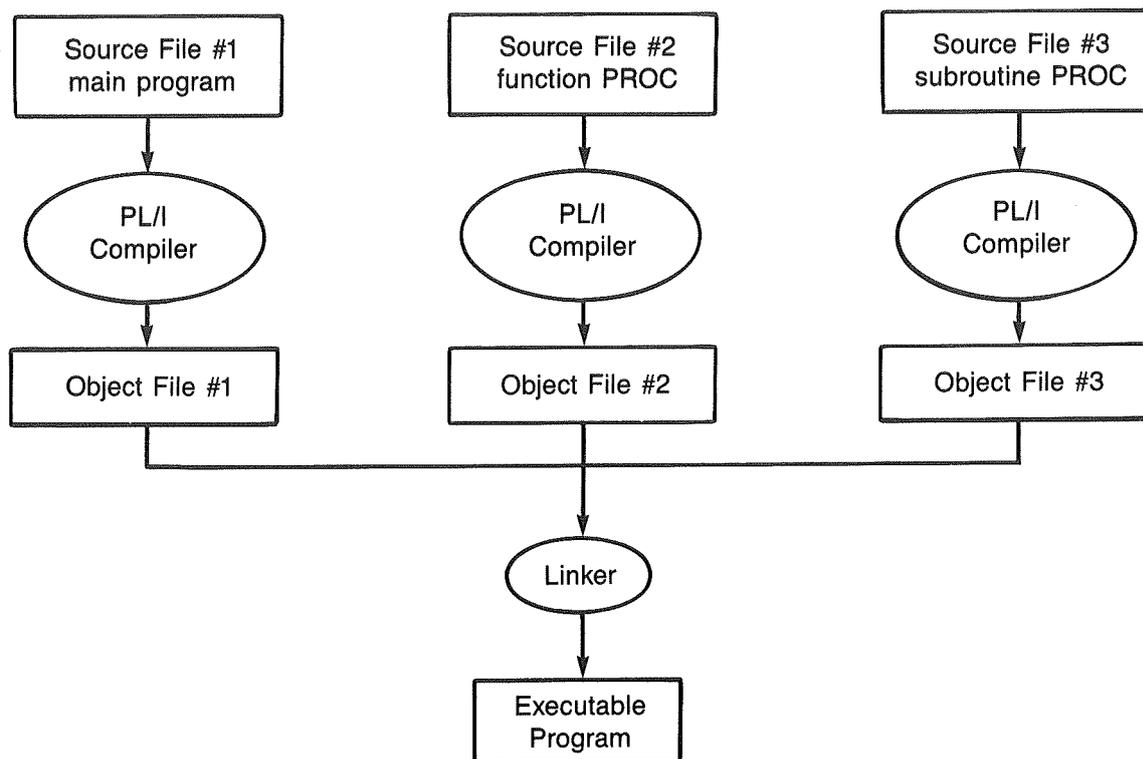
All examples of procedures so far have been internal procedures. An internal procedure is one that is part of another procedure (such as the main program) and that is compiled as part of that procedure. By contrast, an external procedure is separately compiled. Your main program is an example of an external procedure, but it is also possible to have subroutine or function procedures that are all external procedures.

In order to understand the need for external procedures, it is necessary to understand precisely what a compiler is. Figure 8-12 illustrates how a compiler works. Your PL/I program is part of a source file. The source file is used as input to the PL/I compiler, which translates your PL/I program into machine language, storing the result in an object file. Later, the system loads the contents of the object file into the computer's memory, so that the computer can execute the machine language representation of your program.



Compilation
Figure 8-12

When your program contains two or more separately compiled external procedures, the process is a little more complicated. Figure 8-13 illustrates what happens. This figure shows three external procedures, in three separate source files. The first external procedure is the main program, the second is a function procedure, and the third is a subroutine procedure. These external procedures are separately compiled, and the result, as shown in the figure, is three separate object files. In order to get a single executable program from the three object files, you need another system program, called a linker. The linker turns two or more object files into a single executable program. It does this by recognizing and correctly handling the situation where one object file references an external procedure defined in a different object file. The resulting executable program file, or runfile, can be loaded into the computer's memory and executed by the computer.



Compilation of External Procedures
Figure 8-13

Advantages of External Procedures

It is very convenient to include your subroutine and function procedures as internal procedures that are part of your main program.

But a very large program, such as a payroll application or a manufacturing application, can consist of many PL/I statements, perhaps in the millions. Since it is not practical, or even possible, to maintain something so large as a single program, designers of such large programs break them up into smaller chunks, called external procedures. These small external procedures can be written and compiled separately, and then, as a final step, linked together by means of a linker. This method has two major advantages:

- If you change one of the small external procedures, you have no need to recompile the entire system. Instead, simply recompile the one external procedure that you modified, and then link together all object files with the new one.
- If you have many programmers working on the programming project, you can assign different external procedures to different programmers. Each of the programmers can write an external procedure, and can even debug it to some extent, separately. Later, all the external procedures can be linked together to get a single program.

The following sections cover techniques for writing external procedures.

EXTERNAL ENTRY Declarations

External procedures pose certain problems for the compiler that are not present with internal procedures. The problem arises because when the compiler compiles the statement that invokes the procedure, whether a CALL statement or a function reference, the compiler needs certain information about the procedure. In the case of an internal procedure, the internal procedure is compiled at the same time, and so the compiler has complete information about it.

Figure 8-14 illustrates the kind of information that the compiler needs. That example contains an internal procedure, Q, which is invoked by the statement $A = Q(3, B, C)$.

```
P: PROC OPTIONS(MAIN);
  ...
  A = Q(3, B, C);
  ...
Q: PROC(X, Y, Z) RETURNS(FIXED DEC(5));
  DECLARE X FLOAT, Y(5) CHAR(20) VAR, Z BINARY FIXED;
  ...
  END Q;
  END P;
```

Internal Procedure Declaration
Figure 8-14

In order for PL/I to compile this assignment, it needs some information about the procedure Q, which it figures out by simply examining the procedure. Information that it figures out is as follows:

- Q is a function procedure name. It is not a built-in function or an array, which are other possibilities, since in the above statement Q is immediately followed by a parenthesized list.
- Q has three parameters. The aggregate types and data types of these three parameters are scalar FLOAT, array CHARACTER(20) VARYING, and scalar BINARY FIXED, respectively. The compiler needs this information when compiling the above assignment statement in order to know how to handle the three arguments 3, B, and C, and to decide whether or not to create dummy arguments for these arguments.
- Q is a user-defined function that returns a FIXED DECIMAL(5) value. The compiler needs to know this information in order to know what conversions must be done before the value returned by Q(3, B, C) can be assigned to A.

The compiler needs all this information in order to compile a reference to the procedure Q. Now suppose Q is a separately compiled external procedure. PL/I would not have enough information to compile the statement A = Q(3, B, C) because it would not have any of the information in the three paragraphs above.

If you wish your program to be able to invoke a separately compiled external procedure, you must insert a special declaration in your program to give the compiler the information it needs. For example, in Figure 8-14, if Q were not an internal procedure for P, but were a separately compiled external procedure, P would have to contain the following declaration:

```
DECLARE Q EXTERNAL  
ENTRY (FLOAT, (5) CHAR(20) VAR, BIN FIXED)  
RETURNS(FIXED DECIMAL(5));
```

This declaration gives PL/I the information it needs to compile a reference to Q. It provides the following information:

- It says that Q is an EXTERNAL ENTRY. This means that Q is a separately compiled external procedure.
- The ENTRY option is followed by a parenthesized list containing three descriptors. These three descriptors specify that Q has three parameters, and that their aggregate and data types are scalar FLOAT, array CHARACTER(20) VARYING, and BINARY FIXED, respectively. This information allows the compiler to determine whether the arguments in the reference to Q must be replaced by dummy arguments.

SUBROUTINE AND FUNCTION PROCEDURES

- The RETURNS option in the declaration says that Q is a function procedure (not a subroutine procedure), and that the function returns a FIXED DECIMAL(5) value.

Any procedure that calls another external procedure should have one of these declarations. The general format of these declarations is as follows:

```
DECLARE name EXTERNAL
        ENTRY (parameter-descriptor-list)
        [RETURNS (returns-descriptor)];
```

In this format, the parameter-descriptor-list is a list of descriptors separated by commas, giving the data types and aggregate types of each of the procedure parameters.

The RETURNS option must be specified in the declaration if the external procedure is a function, and must be omitted if the external procedure is a subroutine. The returns-descriptor is a descriptor giving the data type and aggregate type of the value returned by the user-defined function.

A returns-descriptor is described in the section Functions That Return an Array or Structure earlier in this chapter. Refer to that discussion for more information on descriptors.

ENTRY Statements and Multiple Entry Points

The label of a PROCEDURE statement, whether the procedure is internal or external, is called the entry point or primary entry point of the procedure. By means of the ENTRY statement, it is possible for you to establish secondary entry points within the procedure. These are additional entry points, which you may use to invoke the procedure and begin execution within the procedure, without having to begin execution at the beginning of the procedure. The following example illustrates the use of the ENTRY statement. There is a main external procedure, P, which contains an internal procedure. The internal procedure has a primary entry point, XSUB, and a secondary entry point, XFNC. XSUB is the label of the PROCEDURE statement, and XFNC is the label of the ENTRY statement. Notice that XSUB is a subroutine entry point, and XFNC is a function entry point to the same procedure. The CALL statement and assignment statement in the main program invoke each of the entry points.

```

P:      PROC OPTIONS(MAIN);
        ...
        CALL XSUB(A);
        ...
        B = XFNC(C);
        ...
XSUB:   PROC(X);
        DECLARE FUNC BIT(1);
        DECLARE(X, Z) FLOAT;
        FUNC = '0' B;
        GO TO COMMON;
XFNC:   ENTRY(X) RETURNS(FLOAT);
        FUNC = '1' B;
COMMON: Z = SQRT(SIN(X) + COS(X));
        IF FUNC THEN RETURN(Z);
        /* RETURN FROM SUBROUTINE XSUB */
        X = Z;
        RETURN;
        END XSUB;
        END P;
    
```

Although the internal procedure is a short one, it uses some techniques that are common to the use of multiple entry points. The real work done inside the procedure, for either entry point, begins at the statement labeled COMMON. In this case, the work done consists of only a single assignment statement, but typical procedures, of course, do much more. Before reaching COMMON, either entry point set the BIT variable FUNC to indicate which entry point was used. That way, the procedure can decide which type of RETURN statement to use. Similar techniques can be used with more than two entry points.

The following general rules apply to multiple entry points:

- Each of the entry points can have different parameters or the same parameters.
- Each of the entry points can be a subroutine or function entry point. Of course, your program must execute the correct form of RETURN statement, depending upon which entry point the caller used. If two or more entry points are function entry points, they can have different RETURNS descriptors.
- Any declaration inside the procedure applies to all of the entry points. In particular, the INITIAL attribute for AUTOMATIC variables is applied, and the declared variables are initialized.
- Multiple entry points can be used with either internal or external procedures. A secondary entry point to an external procedure is also an external entry point, and can be invoked from other external procedures.

- If, during execution of the statements inside a procedure, control reaches an ENTRY statement, then control simply passes around the ENTRY statement.

RECURSIVE PROCEDURES

To invoke a procedure recursively means to invoke it and then to invoke it again while the first invocation is still active. (An invocation remains active until it is terminated by a RETURN or GO TO.)

A recursive invocation can come about either directly or indirectly. It can come about directly if a procedure invokes itself from inside the procedure, as shown in the example below. A recursive invocation can also happen indirectly, as when a procedure invokes a chain of other procedures, one of which invokes the original procedure.

As an illustration of a recursive procedure, consider the factorial function. The formula for this function is as follows:

$$n! = n * (n - 1) * \dots * 2 * 1$$

where $n!$ is read " n factorial." The formula says that n factorial is computed by multiplying together n with all the positive integers smaller than n . For example, $3!$ equals $3 * 2 * 1$ or 6. $6!$ equals $6 * 5 * 4 * 3 * 2 * 1$ or 720. In addition, the special case of $0!$ is defined to equal 1.

Later, we are going to define a recursive procedure that computes the factorial function. However, let us begin with a procedure that is not recursive and that computes n factorial. Such a procedure is shown below. Verify for yourself that FAC1 (3) returns the value 6, FAC1(6) returns 720, and FAC1(0) returns the value 1.

```
FAC1: PROC(N) RETURNS(FIXED);
      DECLARE(K, F) FIXED;
      F = 1;
      DO K = 1 TO N;
        F = F * K;
      END;
      RETURN(F);
      END FAC1;
```

The recursive procedure for n factorial is based on a different definition of the factorial function, a so-called recursive definition. This definition is as follows:

$$0! = 1$$

$$\text{If } n > 0, n! = n * (n - 1)!$$

This definition of n factorial has two lines. The first line specifies what $0!$ is, and the second tells what $n!$ is when $n > 0$. The second line of this definition looks circular, since it seems to define the factorial function in terms of the factorial function. Actually, the definition is not circular, as you can see when we apply the recursive definition to compute the value of $3!$. Applying the second line of the definition, we get

$$3! = 3 * 2!$$

As you can see, we have defined $3!$ in terms of $2!$. This may not seem like the direction in which we wish to go, but we have done something: we have reduced our problem to computing the value of the factorial of a smaller number. If we are able to keep doing that, we will eventually reach $0!$, which we know the value of. We now apply the recursive definition to $2!$ to get

$$3! = 3 * 2!$$

$$= 3 * (2 * 1!)$$

$$= 6 * 1!$$

Applying the second line of the recursive definition to $1!$, we get

$$3! = 6 * 1!$$

$$= 6 * (1 * 0!)$$

$$= 6 * 0!$$

And finally, we apply the first line of the definition, which says that $0!$ equals 1 , to get

$$3! = 6 * 0!$$

$$= 6 * 1$$

$$= 6$$

Similarly, we could compute the value of $n!$ for any positive integer n . We do this by repeatedly applying the second line of the recursive

SUBROUTINE AND FUNCTION PROCEDURES

definition, until we have reduced the problem to $0!$, which the first line of the definition gives us.

The next example contains a recursive procedure that computes the factorial function. This procedure computes factorial in the same way the recursive definition works. The definition clause $0! = 1$ becomes

```
IF N = 0 THEN F = 1;
```

The clause $n! = n * (n - 1)!$ becomes

```
ELSE F = N * FAC2(N - 1);
```

It is this last line that makes FAC2 a recursive procedure. While FAC2 is active, it is possible for it to call itself, so that there are two or more active invocations of FAC2 at the same time.

```
FAC2: PROC(N) RETURNS(FIXED) RECURSIVE;  
  DECLARE F FIXED;  
  IF N = 0 THEN F = 1;  
  ELSE F = N * FAC2(N - 1);  
  RETURN(F);  
END FAC2;
```

If you are going to invoke a subroutine or function procedure recursively, specify the option RECURSIVE on the PROCEDURE statement. Never specify the RECURSIVE option on an ENTRY statement.

Note

On Prime computers, a recursively invoked procedure works properly whether or not the RECURSIVE option is specified.

GENERIC ENTRY NAMES

This is a rarely used capability. It allows you to use a single identifier name to stand for two or more different but related procedures, array names, or built-in functions. Whenever you reference the common identifier name, PL/I uses the data types and aggregate types of the arguments to determine which of the procedures, arrays, or built-in functions to choose.

Here are some examples:

- `DECLARE Q GENERIC (Q1 WHEN (*), Q2 WHEN (*,*), Q3 WHEN (*,*,*));`

This `DECLARE` statement might be used in a program that contains three different but related procedures, `Q1`, `Q2`, and `Q3`. The program may reference `Q` as if it were a procedure name, and then PL/I replaces the reference to `Q` with a reference to `Q1`, `Q2`, or `Q3`, depending upon the arguments you specify in the referenced `Q`. In the declaration `Q` shown above, the words `Q1 WHEN(*)` indicate that PL/I is to use `Q1` for `Q` whenever the reference to `Q` contains exactly one argument. The words `Q2 WHEN(*,*)` indicate that `Q2` is to be substituted for `Q` when the reference to `Q` has exactly two arguments. And the words `Q3 WHEN(*,*,*)` specify that `Q3` is to be used when the reference to `Q` contains exactly three arguments.

Therefore, PL/I would change a reference to `Q(X)` to a reference to `Q1(X)`, and would change a reference to `Q(X, Y + 3)` to a reference to `Q2(X, Y + 3)`. Therefore, the statement

```
Z = Q(X) * Q(X, Y + 3);
```

would be equivalent to

```
Z = Q1(X) * Q2(X, Y + 3);
```

- `DECLARE LN_ARRAY (5) FLOAT INIT(1, 2.7182818, 7.3890561,
20.0855369, 54.5981500);`

`DECLARE LN GENERIC (LN_ARRAY WHEN (FIXED (1:31, 0)),
LOG WHEN (*));`

In the `GENERIC` declaration, the phrase `FIXED(1:31, 0)` refers to any `FIXED` data type with a precision containing one to 31 digits, none of which follow the decimal (or binary) point. Therefore, this is a way of representing an integer data type in a `GENERIC` declaration.

The `GENERIC` declaration specifies that any reference to `LN` is to be replaced with a reference to either the array `LN_ARRAY` or the built-in function `LOG`. If the argument to `LN` has one of the integer data types just described, `LN` is replaced with `LN_ARRAY`; but if an argument of any other data type is used, the built-in function `LOG` replaces `LN`.

A programmer might use a declaration of this kind to improve performance of a program. Since an array reference is presumably faster than a built-in function reference, the program references the array whenever the argument is an

SUBROUTINE AND FUNCTION PROCEDURES

integer, and invokes the built-in function whenever the argument is not an integer. Of course, the usefulness is rather limited, since the generic choice is based not on whether the value of the argument is an integer but rather on whether the data type of the argument is integer. Furthermore, the program gets an execution error if the argument to LN has an integer data type whose value is not in the range 1 to 5. For example, if I has the attribute FIXED DECIMAL(5), PL/I replaces a reference to LN(I) with a reference to LN_ARRAY(I), and this reference is illegal if the value of I is less than 1 or greater than 5.

The format of a GENERIC declaration is as follows:

```
DECLARE ident GENERIC(choice, choice, ...);
```

where each choice is of the form

```
name WHEN(descriptor, descriptor, ...)
```

When your program references ident, PL/I replaces the reference to ident with a reference to one of the choices of name, based on a process of matching the arguments in the reference to ident with the list of descriptors in each of the choices in the manner described below.

If descriptor corresponds to one argument, it gives the range of data types and aggregate types in the argument that can be used for this particular choice of name. If the data types and aggregate types of all the arguments in the reference to the ident satisfy the specifications of the descriptors for a choice in the GENERIC declaration, then the ident is replaced by the name for that choice. If the argument list satisfies the descriptor specifications for more than one choice in the GENERIC declaration, the first choice is used.

The descriptor for a single argument may be a structure descriptor like

```
1, 2 CHARACTER, 2 FIXED;
```

This descriptor would match an argument that was a structure containing two members, the first of which is CHARACTER and the second of which is FIXED.

The descriptor may also include array bounds, specifying that the corresponding argument must be an array. In this case, you must specify an asterisk for the value of the array bound. For example, the descriptor

(*) FLOAT DECIMAL

matches any singly-dimensioned array that is FLOAT DECIMAL.

Each descriptor can specify any of the following attributes:

ALIGNED	LABEL
AREA	MEMBER
BINARY [(precision)]	NONVARYING
BIT	OFFSET
CHARACTER	PICTURE 'picture specification'
COMPLEX [(precision)]	POINTER
DECIMAL [(precision)]	PRECISION [(precision)]
DIMENSION(*, ...)	REAL [(precision)]
ENTRY [[descriptor, ...]]	RETURNS [(descriptor)]
FILE	STRUCTURE
FIXED [precision]	UNALIGNED
FLOAT [precision]	VARYING
FORMAT	

In each case in the above list where precision is specified, you may specify either the number of digits or the number of digits and the scale factor. Furthermore, you may specify either quantity as a range by using the colon (:). For example, the descriptor

FIXED DECIMAL(1:10, -5:0)

refers to any FIXED DECIMAL data type with a precision of between one and 10 digits, with a scale factor between -5 and 0.

ENTRY VARIABLES

The ENTRY data type is a noncomputational data type. The label of a PROCEDURE or ENTRY statement is an identifier to which PL/I gives the ENTRY data type. Such a statement label is considered to be a constant of the ENTRY data type.

By means of an appropriate DECLARE statement, you may specify that an identifier of your choice is to be an ENTRY variable. You may assign ENTRY values, such as ENTRY constants or other ENTRY variables, to an

SUBROUTINE AND FUNCTION PROCEDURES

ENTRY variable, but you may not perform ordinary arithmetic computations or ordinary input or output operations on such variables.

The following example illustrates a simple use of ENTRY variables. In this program segment, the DECLARE statement specifies that EV is to be an ENTRY variable. The IF statement tests the variable K to decide which ENTRY constant, P or Q, to assign to the ENTRY variable EV. The CALL statement that follows specifies the ENTRY variable EV; the actual procedure that is invoked by this CALL statement depends upon the current value of EV, either P or Q.

```
DECLARE EV VARIABLE ENTRY;
...
IF K = 1 THEN EV = P;
  ELSE EV = Q;
...
CALL EV;
...
P: PROC;
...
END P;
Q: PROC;
...
END Q;
```

An ENTRY variable may be part of a structure or may be an array. Arrays of ENTRY variables can provide a table-driven programming capability. The next example illustrates this. In this case, the ENTRY constants P, Q, and R are external procedures rather than internal procedures. Each of these external procedures has a single scalar FLOAT parameter and is a function procedure that returns a scalar FLOAT value.

```
DECLARE(P, Q, R) EXTERNAL
  ENTRY(FLOAT) RETURNS(FLOAT);
DECLARE F(3) VARIABLE
  ENTRY(FLOAT) RETURNS(FLOAT)
  INITIAL(P, Q, R);
...
X = F(K)(Y);
```

In this example, F is an array of three ENTRY variables. The INITIAL attribute in the declaration of F specifies that F(1) is to be initialized to the ENTRY constant P, F(2) is to be initialized to Q, and F(3) is to be initialized to R. The declaration for F must also specify the parameter-descriptors and the returns-descriptor, for the same reasons that these descriptors must be specified for external ENTRY constants. The PL/I compiler needs this information in order to be able to decide whether to create dummy arguments and whether a conversion of a return value is needed.

Notice the surprising assignment statement

```
X = F(K) (Y);
```

This statement is valid only when K has the value 1, 2, or 3. F is an array ENTRY variable, and F(K) has, as its value, one of the ENTRY values P, Q, or R. Therefore, depending upon the value of K, this assignment statement is equivalent to one of the three following:

```
X = P(Y);  
X = Q(Y);  
X = R(Y);
```

Which of P, Q, or R is invoked depends upon the value of K.

ADVANCED PROGRAMMING OPTIONS: SHORICALL AND NONQUICK

Prime PL/I offers two options, SHORICALL and NONQUICK, for programmers who wish to have greater control over the procedure-calling mechanism.

If a PL/I program calls an external procedure written in PMA (Prime Macro Assembler), you may declare that procedure with the SHORICALL option, using the following syntax:

```
DCL EXPROC EXTERNAL ENTRY OPTIONS(SHORICALL);
```

The SHORICALL option generates a JSXB rather than a PCL sequence for invocations of this procedure. You should be aware of the following:

- You must code the PMA subroutine to use the runtime support scratch space in the stack. See Appendix D, Figure D-1, for the location of this scratch space.
- Parameters for a SHORICALL subroutine are passed by reference, and you must code the subroutine to handle them properly. If the subroutine takes only one argument, the address of the argument is placed in the L-register. If it takes more than one, the SHORICALL option creates an array of pointer temporaries, puts the address of each argument in that array, and puts the address of the array into the L-register.
- If only one argument is passed to a SHORICALL routine, it must be word-aligned. If the routine takes multiple arguments, their alignment does not matter.

SUBROUTINE AND FUNCTION PROCEDURES

The second option, `NONQUICK`, applies to internal rather than external procedures. If a program is to be compiled at an optimization level of 3 or above, the compiler ordinarily makes most non-recursive internal procedures `SHORTCALL`. If you wish to prevent this from happening to a particular internal procedure, use the following syntax:

```
INPROC: PROCEDURE OPTIONS(NONQUICK);
```

The procedure is then called with the `PCL` rather than the `JSXB` sequence.

SUMMARY OF PROCEDURE RULES

This section summarizes some of the major rules in using procedures.

- The main procedure of your program begins with the statement

```
name: PROCEDURE OPTIONS(MAIN);
```

- The abbreviation for `PROCEDURE` is `PROC`.
- A subroutine procedure begins with a statement in the format

```
name: PROCEDURE [(pam-list)] [RECURSIVE];
```

The pam-list is the list of parameters for the procedure, and is written as a list of one or more identifiers separated by commas.

- Any declarations inside a procedure apply only within that procedure. In particular, you may specify the data type of a parameter by using a `DECLARE` statement within the procedure to specify any attributes that you wish the parameter to have.
- A function procedure begins with a statement in the format

```
name: PROCEDURE [(pam-list)] [RECURSIVE]  
      RETURNS [(descriptor)];
```

The descriptor specifies the data type and aggregate type of the value to be returned by the function procedure.

- Invoke a subroutine procedure with a statement in the form

```
CALL name [(arg-list)];
```

- Invoke a function procedure by using a reference of the form

```
name [(arg-list)]
```

in an expression.

The arg-list is the list of arguments to be passed when invoking the procedure. The list of arguments is one or more expressions separated by commas. The number of arguments used when invoking a procedure must equal the number of parameters in the PROCEDURE statement.

- During execution of the invoked procedure, each of the procedure parameters points back to the corresponding argument used in invoking the procedure. In certain cases, PL/I creates a dummy argument when invoking the procedure, and then the parameter points back to the dummy argument. The cases where a dummy argument is created are as follows:
 - When the argument is a constant.
 - When the argument is an expression (as when it is enclosed in a set of parentheses).
 - When the data type or aggregate type of the argument is different from that of the parameter.

In order to determine whether a dummy argument is needed, the compiler must know, at the time it is compiling the statement that invokes a procedure, what the data types of the procedure parameters are. If the procedure being invoked is an internal procedure, the compiler can determine the data types of the parameters. However, if the procedure being invoked is an external procedure that is separately compiled, the invoking program must contain a special declaration to indicate to the compiler what the data types of the parameters are. The format of this declaration is as follows:

```
DECLARE name EXTERNAL ENTRY (parm-descriptors)  
      [RETURNS(returns-descriptor)];
```

The parm-descriptors are the descriptors for each of the parameters of the procedure, separated by commas. There is one descriptor for each parameter, and it specifies the data type

SUBROUTINE AND FUNCTION PROCEDURES

and aggregate type of the parameter. Similarly, the returns-descriptor specifies the data type and aggregate type of the value returned by the procedure, if it is a function procedure.

- To return from a subroutine procedure, your program must execute the statement

```
RETURN;
```

or else the END statement of the procedure. To return from a function procedure, your program must execute a statement of the format

```
RETURN(expression);
```

It is illegal to execute the END statement of a procedure invoked as a function.

- A procedure may have one or more secondary entry points. Each secondary entry point is specified by the name of an ENTRY statement in the following format:

```
name: ENTRY [(parm-list)] [RETURNS [(descriptor)]];
```

9

Program Blocks, Declarations, and Scope Rules

PL/I PROGRAM BLOCK STRUCTURES

PL/I is a so-called block structured language, meaning that a PL/I program consists of a group of nested blocks. This feature, described in the following pages, gives the PL/I programmer a great deal of power.

Statement Groups and Blocks

Three statement types require a matching END statement: the DO, PROCEDURE, and BEGIN statements. Each of these statements, along with its corresponding END statement, identifies a collection of statements to be handled in a special way.

For example, the entire compiled program is a collection of statements beginning with a PROCEDURE statement and ending with an END statement. This is called an external procedure. Within the external procedure, there may be additional PROCEDURE, BEGIN, and DO statements, each matched with its own END statement. Depending upon the circumstances, the collection of statements so defined might be a subroutine, a user-defined function, a group of statements to be repeated, or an error-handling routine.

Figure 9-1 illustrates the three statement types. In this example, P is an external procedure, with the initial PROCEDURE statement matched by the final END statement. Inside the procedure P, there are

- A DO group. In this case, the group is a group of statements to be executed iteratively. The DO statement is covered in Chapter 10.
- Three internal blocks, each beginning with either PROCEDURE or BEGIN, and ending with END.

According to PL/I terminology, DO and END define a group of statements, while BEGIN and END or PROCEDURE and END define a block of statements.

This chapter deals with blocks. We mention the DO group in this chapter only because both the group and the block require an END statement. The full use of the DO statement is described in Chapter 10.

```
P: PROCEDURE;  
  ...  
  DO K = 1 TO 5;  
    ...  
    END;  
    ...  
    BEGIN;  
    ...  
    END;  
    ...  
    ON ERROR BEGIN;  
    ...  
    END;  
Q: PROC;  
  ...  
  END Q;  
  END P;
```

Statement Types
Figure 9-1

PL/I's block structure capability gives the user a great deal of power in the following areas:

- Limiting the scope of a declaration: that is, you can specify the range of statements over which the declaration applies.

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

- Modular programming: using the procedure capability to break up a large program into small chunks. See Chapter 8 for details.
- Error and condition handling: using the ON-unit to define what happens when a condition or error occurs. See Chapter 13.
- Recursive programming, the ability to have more than one simultaneous activation of a block: this can be very powerful in certain computer programs, such as compilers. See Chapter 8.

Only blocks (as opposed to DO groups) are relevant to these points. The DO group provides for repetitive execution of a collection of statements, but does not provide the kind of control described above for blocks.

Types of Blocks

PL/I syntax recognizes three types of blocks, each of which begins with either a PROCEDURE statement or a BEGIN statement:

- A PROCEDURE block, which begins with a PROCEDURE statement and ends with an END statement. Use a PROCEDURE block to define your main program or to create a subroutine or user-defined function. Such procedures may be internal or external. Procedures are defined in Chapter 8.
- An ON-unit. This is a BEGIN block that is attached to an ON statement. It is used for trapping errors or other conditions that might make a program abort. The format is

```
ON condition-name [SNAP] BEGIN;  
  ...  
END;
```

PL/I handles ON-units differently from the way it handles BEGIN blocks that are not ON-units. Use ON-units for error and condition handling, as described in Chapter 13.

- A BEGIN block other than an ON-unit. Such a block provides block structuring without the additional features provided by PROCEDURE blocks or ON-units (subroutines and functions, and error and condition handling).

Nesting of Blocks

A PL/I compiler compiles a module called an external procedure. This external procedure may be a main program, a subroutine, or a function. Inside the external procedure may be one or more internal blocks, PROCEDURE blocks, BEGIN blocks, and ON-units. Each of these internal blocks may have any other internal blocks inside it.

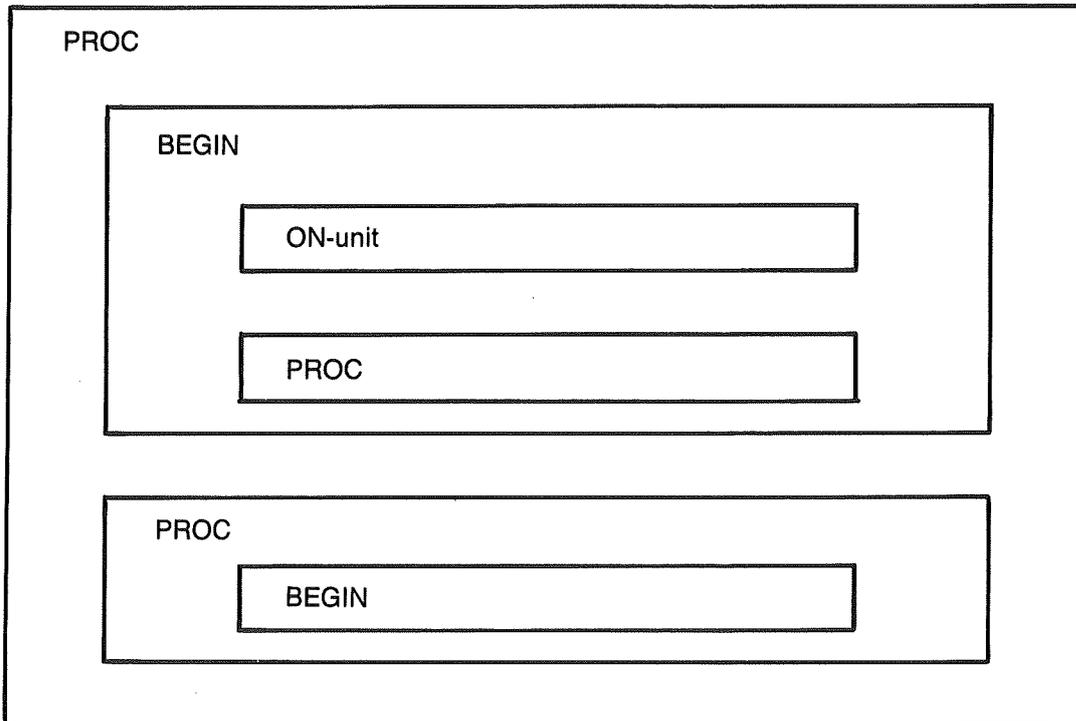
Figure 9-2 shows a program skeleton with several nested blocks of various types. Compare this program with Figure 9-3. By comparing the program with its representation, you can see the meaning of the block nesting concept. An external procedure (the entire main program) has five internal blocks. An internal BEGIN block has an ON-unit and a PROCEDURE block inside of it. A PROCEDURE block has a BEGIN block nested inside.

```

P:  PROCEDURE OPTIONS(MAIN);
    BEGIN;
        ...
        ON ERROR BEGIN;
            ...
            END;
Q:  PROC;
        ...
        END Q;
    END;
R:  PROC;
    BEGIN;
        ...
        END;
        ...
    END R;
END P;

```

Block Structure
Figure 9-2



Representation of Block Structure
Figure 9-3

Multiple Closure END Statements

Each DO, BEGIN, and PROCEDURE statement in your program must have a corresponding END statement. If you require several END statements in a row, PL/I permits you to use a single END statement in the format

END identifier;

If you use an END statement in this syntax, PL/I supplies additional END statements, to close off any unclosed groups or blocks, back to the DO, PROCEDURE, or BEGIN statement whose label is the specified identifier.

Consider this program segment, which has a DO group, a BEGIN block, and a PROCEDURE block, all ending at the same point.

```
L: DO;
    ...
M: BEGIN;
    ...
N: PROCEDURE;
    ...
    END L;
```

As a result of the statement END L, PL/I inserts two additional END statements just before this statement, to close off the BEGIN block and the PROCEDURE block. The result is as shown below. Of course, these END statements do not appear in your program listings, since the compiler inserts them internally without displaying the result.

```
L: DO;
    ...
M: BEGIN;
    ...
N: PROCEDURE;
    ...
    END;
    END;
    END L;
```

THE DECLARE STATEMENT

Use the DECLARE statement to give identifiers the attributes of your choice. For example, the DECLARE statement

```
DECLARE X FIXED BINARY STATIC;
```

specifies that the identifier X is to have the attributes FIXED, BINARY, and STATIC.

The DECLARE statement has many formats. The simplest format is

```
DECLARE identifier [attribute-list];
```

which specifies that the identifier is to have all the attributes in the attribute-list. The attribute list may include some combination of the keywords FIXED, FLOAT, DECIMAL, BINARY, COMPLEX, REAL, INITIAL, and PICTURE discussed in Chapter 5.

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

The left and right brackets in the above format indicate that the attribute-list is optional, and so the DECLARE statement can be used to declare an identifier without specifying any attributes. In this case, the identifier is given the default attributes. For example, the statement

```
DECLARE Z;
```

declares Z to have the system default attributes, BINARY FIXED.

Declaring Structure Level Numbers

When you use the DECLARE statement to declare a structure, specify level numbers for the member names. In the following example, the structure name EMPLOYEE has a level number of 1. EMPLOYEE has three members, NAME, SALARY, and STARTDATE, each at level 2. SALARY and STARTDATE are substructures, each with level 3 members.

```
DECLARE 1 EMPLOYEE(1000),  
        2 NAME CHARACTER(20) VARYING,  
        2 SALARY,  
          3 REGULAR PICTURE '$$$V.99',  
          3 OVERTIME PICTURE '$$$V.99',  
        2 STARTDATE,  
          3 MONTH FIXED DECIMAL(2),  
          3 DAY FIXED DECIMAL(2),  
          3 YEAR FIXED DECIMAL(4);
```

Your structure declaration need not have consecutive level numbers. The next example illustrates this.

```
DECLARE 1 EMPLOYEE(1000) ,  
      3 NAME CHARACTER(20) VARYING,  
      3 SALARY,  
      7 REGULAR PICTURE '$$$V.99',  
      7 OVERTIME PICTURE '$$$V.99',  
      3 STARTDATE,  
      6 MONTH FIXED DECIMAL(2) ,  
      6 DAY FIXED DECIMAL(2) ,  
      6 YEAR FIXED DECIMAL(4);
```

This declaration has exactly the same effect as the one before it, but the level numbers are different. Internally, when PL/I compiles your program, PL/I changes your level numbers to the correct logical level numbers. In the case of the declaration in the example below, the logical level numbers are those in the preceding example. Of course, PL/I changes the level numbers only internally; the level numbers you specify appear in your listing.

Any declared identifier that is not a structure member has a level number of 1, even if it is not a structure. If you declare an identifier with no level number, PL/I assumes a level number of 1. Consider the declarations below. In these examples, X, Y, S, and T are level-1 identifiers. Note that X and Y are level-1 scalars, while S and T are level-1 structures. X and Y are considered level-1 even though they are not structures. A, B, C, and D are level-2 identifiers that are members of structures.

```
DECLARE X FLOAT;  
DECLARE 1 Y FLOAT;  
  
DECLARE 1 S, 2 A, 2 B;  
DECLARE 1 T, 2 C, 2 D;
```

Multiple Declarations

You may use a single DECLARE statement to declare two or more identifiers. For example, the single DECLARE statement

```
DECLARE A FLOAT, B FIXED,  
      C(20) CHARACTER(100) VARYING;
```

is equivalent to the following three DECLARE statements:

```
DECLARE A FLOAT;  
DECLARE B FIXED;  
DECLARE C(20) CHARACTER(100) VARYING;
```

In both cases, you are giving the identifiers A, B, and C exactly the same attributes.

Factored Declarations

Many times when you are writing a program, you would like to give several different identifiers the same or similar attributes. In such cases, the technique of factoring declarations can greatly simplify your DECLARE statements. Some examples follow.

If you wish to give several identifiers exactly the same attributes, follow a parenthesized list of the identifiers with a list of the common attributes. For example,

```
DECLARE (A, B, C) CHAR(20) VAR;
```

gives A, B, and C the same attributes, CHAR(20) VAR, and so is the same as

```
DECLARE A CHAR(20) VAR, B CHAR(20) VAR,  
        C CHAR(20) VAR;
```

which does the same thing.

WARNING

A very troublesome error for beginners is to forget to parenthesize the list of identifiers. For example, many beginners would accidentally write the factored DECLARE statement above as

```
DECLARE A, B, C CHAR(20) VAR;
```

The reason this is such a problem is that the DECLARE statement without parentheses is completely legal, and so is not flagged by the PL/I compiler, and yet it gives results that are totally unexpected by the user. The DECLARE statement just above would give A and B the default attributes (BINARY FIXED), and would give C the attributes of CHAR(20) VAR, instead of, as the user expects, giving all three identifiers the attributes CHAR(20) VAR.

The factoring technique may be used even when the identifiers being declared do not have all of their attributes in common, as long as they share some attributes. In this case, it is possible to factor out just the attributes that they do have in common. For example, the statement

```
DECLARE (M BINARY, N DECIMAL(5),  
        P DECIMAL(7)) FIXED;
```

is equivalent to the following three statements:

```
DECLARE M BINARY FIXED;  
DECLARE N DECIMAL(5) FIXED;  
DECLARE P DECIMAL(7) FIXED;
```

Here the parenthesized list contains not only the identifiers M, N, and P being declared, but also the attributes that these identifiers do not share (BINARY, DECIMAL(5), and DECIMAL(7)). The only factored attribute is the shared attribute FIXED.

As another example, consider the following:

```
DECLARE (X FLOAT, Y DEC(2) FIXED)  
        STATIC COMPLEX;
```

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

The two factored attributes here are `STATIC` and `COMPLEX`. Therefore, the two `DECLARE` statements

```
DECLARE X FLOAT STATIC COMPLEX;  
DECLARE Y DEC(2) FIXED STATIC COMPLEX;
```

are equivalent to the one statement above.

If you wish to make several identifiers into an array, you may factor out the parenthesized dimension list. For example, the statement

```
DECLARE (LA, IB) (10) FIXED;
```

factors out the dimension specification, 10. Therefore, the statement

```
DECLARE LA(10) FIXED, IB(10) FIXED;
```

is equivalent.

Whenever a parenthesized list immediately follows the right parenthesis of a factorization, PL/I treats the parenthesized list as a dimension list. For example, the declaration

```
DECLARE (A BIN, B DEC) (5,7);
```

factors out the dimension list (5,7). An equivalent declaration is

```
DECLARE A(5,7) BIN, B(5,7) DEC;
```

The (5,7) is treated as a dimension list, not as a precision to be associated with `BINARY` and `DECIMAL`.

You can also use several nested levels of factoring in your DECLARE statement. For example, the statement

```
DECLARE ((F VAR, G) CHAR(20), H DECIMAL)
        STATIC;
```

contains two nested levels of factoring, with three identifiers, F, G, and H, being declared. This factored DECLARE statement is equivalent to the following three DECLARE statements:

```
DECLARE F VAR CHAR(20) STATIC;
DECLARE G CHAR(20) STATIC;
DECLARE H DECIMAL STATIC;
```

You may also use the factoring technique to factor an entire structure. For example, the declaration

```
DECLARE 1 (S, T), 2 A FIXED, 2 B FLOAT;
```

specifies that the two structures S and T are to have the same members. Therefore, it is equivalent to the declarations

```
DECLARE 1 S, 2 A FIXED, 2 B FLOAT;
DECLARE 1 T, 2 A FIXED, 2 B FLOAT;
```

You may also factor structure members inside a structured declaration. For example, the declaration

```
DECLARE 1 ST, 2 (A, B) FLOAT;
```

defines a structure, ST, with two members, A and B, each of which is FLOAT. Therefore, the declaration

```
DECLARE 1 ST, 2 A FLOAT, 2 B FLOAT;
```

is equivalent.

The LIKE Attribute

Use the LIKE attribute when you wish to declare a structure to have the same members as one declared elsewhere. In the following declarations,

```
DECLARE 1 S, 2 A FIXED, 2 B FLOAT;
DECLARE 1 T LIKE S;
```

the second declaration specifies that T is a structure that is to have the same members as S. Therefore, the declaration

```
DECLARE 1 T, 2 A FIXED, 2 B FLOAT;
```

is an equivalent declaration for T.

The LIKE attribute can be used to copy the declarations of structure members and their attributes from one structure declaration to another. It does not copy storage type or dimension attributes for the structure identifier itself. For example, the declarations

```
DECLARE 1 ST(20) AUTOMATIC, 2 X, 2 Y;
DECLARE 1 STB BASED LIKE ST;
```

give STB the same members (X and Y) as the array of structures ST. However, the dimension attribute, 20, and the storage class AUTOMATIC, are not copied to STB. In fact, STB is a BASED scalar. Therefore,

```
DECLARE 1 STB BASED, 2 X, 2 Y;
```

is an equivalent definition for the structure STB.

TYPES OF DECLARATIONS

All identifiers (other than keywords) that appear in your PL/I program are declared in one way or another. If you do not explicitly declare such an identifier in one of the three ways below, PL/I supplies either a contextual or an implicit declaration of its own. The rules for explicit and nonexplicit declarations are described in this section.

Types of Explicit Declarations

PL/I recognizes three contexts that explicitly declare an identifier. These contexts, and the resulting attributes that PL/I gives the identifier, are as follows:

- An identifier declared by a DECLARE statement is explicitly declared to have the attributes specified in the DECLARE statement.
- An identifier appearing as a statement label is explicitly declared to be a named constant. The precise attributes that PL/I gives to such an identifier depend upon the kind of statement being labeled. If it is a FORMAT statement, the identifier is given the attributes FORMAT CONSTANT. If the statement is a PROCEDURE or ENTRY statement, the identifier is given the attributes ENTRY CONSTANT. In all other cases, the identifier is given the attributes LABEL CONSTANT. In the case of an ENTRY constant, if the PROCEDURE statement is an external procedure, or if the ENTRY statement is for a secondary entry point for an external procedure, then the identifier is given the EXTERNAL attribute; otherwise, it is given the INTERNAL attribute.
- An identifier appearing in the context of a parameter in a parameter list of a PROCEDURE or ENTRY statement is thereby explicitly declared to have the PARAMETER attribute.

Figure 9-4 is a program skeleton containing a number of explicit declarations. The explicit declarations are as follows:

- Since P is a statement label for a PROCEDURE statement for an external procedure, P has the attributes EXTERNAL ENTRY.
- X, by virtue of its appearance in the DECLARE statement, is explicitly declared to have the attribute FLOAT.
- L is a statement label for a statement other than a PROCEDURE, ENTRY, or FORMAT statement. Therefore, L is explicitly declared to have the attributes LABEL CONSTANT.
- FR is a statement label for a FORMAT statement, and so is explicitly declared to have the attributes FORMAT CONSTANT discussed in Chapter 11.
- Of the three explicit declarations for A in the internal procedure, two are in the context of a procedure parameter, and one is in a DECLARE statement. As a result of these three explicit declarations, A receives the attributes CHARACTER(20) PARAMETER.

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

- B is explicitly declared to be a `PARAMETER`, because it appears in the parameter list of the `ENTRY` statement. Since there is no separate `DECLARE` statement for B, the data type attributes for B are the system or program defaults.
- SB appears as the label of a `PROCEDURE` statement for an internal procedure, and so SB receives the attributes `INTERNAL ENTRY (CHARACTER(20)) CONSTANT`.
- FN appears as the statement label for an `ENTRY` statement for an internal procedure, and thereby receives the attributes `INTERNAL ENTRY (CHARACTER(20), BINARY FIXED) RETURNS(FLOAT) CONSTANT`.

```
P:  PROC OPTIONS(MAIN);
    DECLARE X FLOAT;
    ...
L:  PUT EDIT(X) (R(FR));
FR: FORMAT(F(5));
    ...
SB: PROCEDURE(A);
    DECLARE A CHARACTER(20);
    ...
FN: ENTRY (A,B) RETURNS(FLOAT);
    ...
    END SB;
    END P;
```

Explicit and Non-explicit Declarations
Figure 9-4

LABEL CONSTANT Arrays: An identifier that appears in the context of a statement label is thereby explicitly declared by such an appearance. You may put subscripts on such identifier references to create a `LABEL CONSTANT` array.

The next example illustrates the use of such an array. In this example, the appearances of `BRANCH` in statement labels cause `BRANCH` to be explicitly declared as a `LABEL CONSTANT` array, with a lower bound of 1 and an upper bound of 10. PL/I decides the upper and lower bounds by finding the lowest and highest subscripts, respectively, that appear in the statement label references.

```
      .  
      GO TO BRANCH(K);  
      .  
BRANCH(1): ...  
      .  
BRANCH(2): ...  
      .  
BRANCH(3): ...  
      .  
      .  
BRANCH(10): ...
```

If you use this method to declare LABEL CONSTANT arrays, all appearances of the identifier in the context of a label must have the same number of subscripts, and all subscripts must be constants. Furthermore, the various subscript lists must all be different.

Contextual Declarations

Some other kinds of declarations are not explicit in the sense we have described.

When you use an identifier in an explicit declaration, you do so for the express purpose of telling the PL/I compiler what attributes the identifier should have. This is true whether the explicit declaration is performed by means of a DECLARE statement, a statement label, or a PROCEDURE or ENTRY statement parameter list.

If you use an identifier (not a keyword) in your program, and there is no explicit declaration for that identifier, then PL/I must supply a declaration for you. In most cases, PL/I supplies an implicit declaration, as described in the next section. However, if you happen to use that identifier in your program in certain contexts, PL/I makes a contextual declaration of the identifier, giving the identifier certain attributes, depending upon the context.

The following paragraphs define these contexts and the attributes derived from the resulting contextual declarations.

Built-in Functions: The most common contextual declaration is for built-in functions. Unless you explicitly declare the built-in function identifier name to have the BUILTIN attribute, PL/I has no way of knowing whether the identifier is to be an ordinary variable or a built-in function. However, if the identifier appears in some statement of your program in the context of being immediately followed by a parenthesized argument list, PL/I assumes that it is a built-in function and gives it the BUILTIN attribute in a contextual declaration. For example, if

```
A = MAX(B,C);
```

appears in your program, and if there is no explicit declaration of MAX as either an ENTRY or an array, then PL/I assumes that MAX is a built-in function. PL/I contextually declares the identifier MAX to have the BUILTIN attribute.

On the other hand, it is perfectly legal to use a built-in function name as an ordinary variable in your program. For example,

```
MIN = X + Y;
```

is a valid assignment statement to an ordinary variable called MIN, provided that you do not use MIN as a built-in function elsewhere in your program. If you do, MIN would be given the BUILTIN attribute, and the above assignment statement would then be illegal.

It is important to understand the concept of contextual declarations when you are using built-in functions that take no arguments, such as DATE, TIME, and ONSOURCE. For example, if you use DATE in your program in the statement

```
PUT LIST(DATE);
```

then, since DATE is not followed by a parenthesized argument list, PL/I does not contextually declare DATE to be BUILTIN. As a result, DATE is considered an ordinary variable, and the above PUT statement prints some numeric value, usually 0. To make the above statement work properly, follow DATE with a pair of empty parentheses, as in

```
PUT LIST(DATE());
```

The empty argument list following the identifier DATE causes PL/I to make the correct contextual declaration. If you use the DATE built-in function many times in your program, you only need to put the empty argument list after one occurrence. The contextual declaration then

applies to all occurrences. Of course, you could avoid the whole issue by using an explicit declaration like

```
DECLARE DATE BUILTIN;
```

This would be an explicit declaration of DATE as BUILTIN, and so no contextual declaration would be needed.

File Constant Identifiers: Another common contextual declaration is for FILE CONSTANT identifiers. For example, if your program contains the statement

```
READ FILE(TAPEIN) INTO(REC);
```

and if your program has no explicit declaration for the identifier TAPEIN, then PL/I contextually declares TAPEIN to have the FILE CONSTANT attributes.

There are several contexts that would give an identifier the FILE CONSTANT attributes in a contextual declaration. These are as follows:

- FILE(identifier) option in the OPEN, PUT, GET, READ, WRITE, REWRITE, DELETE, or CLOSE statement.
- COPY(identifier) option in the PUT statement.
- One of the file condition options in the ON, REVERT, or SIGNAL statement. The file conditions are ENDFILE(identifier), UNDEFINEDFILE(identifier), ENDPAGE(identifier), KEY(identifier), NAME(identifier), RECORD(identifier), and TRANSMIT(identifier).

Of course, if your program contains an explicit declaration for the identifier -- for instance, giving it the FILE VARIABLE attributes -- then no contextual declaration is made.

An interesting special case of FILE CONSTANT contextual declarations occurs for SYSIN and SYSERIN. For example, if your program contains a GET statement with no FILE or STRING option, FILE(SYSIN) is assumed by PL/I. As a result of this assumption, PL/I contextually declares SYSIN to be FILE CONSTANT. Therefore, the statement

```
GET LIST(X);
```

is assumed by PL/I to mean the same thing as

```
GET FILE(SYSIN) LIST(X);
```

and so `SYSIN` is contextually declared as a `FILE CONSTANT`. A similar thing happens with `SYSPRINT`. If no `FILE` or `STRING` option occurs in a `PUT` statement, PL/I assumes the `FILE(SYSPRINT)` option for the `PUT` statement and contextually declares `SYSPRINT` to be a `FILE CONSTANT`.

The Condition Attribute: If an identifier appears with the `CONDITION` option of the `ON`, `REVERT`, or `SIGNAL` statement, PL/I contextually declares that identifier to have the `CONDITION` attribute. For example, the statement

```
ON CONDITION(FILEERROR) GO TO HANDLE;
```

causes PL/I to declare `FILEERROR` contextually to have the `CONDITION` attribute. See Chapter 13 for a full discussion of PL/I condition handling.

The POINTER Attribute: In certain contexts, an undeclared variable is given the `POINTER` attribute. These contexts are as follows:

- If the identifier appears to the left of the symbol `->`. For example, suppose your program contains the statement

```
P->S = 15
```

If `P` is not explicitly declared, then, as a result of this statement, PL/I contextually declares `P` to have the `POINTER` attribute.

- If the identifier appears in the `SET` option of the `READ` or `ALLOCATE` statement. For example, if `P` is undeclared, the statement

```
READ FILE(TAPEIN) SET(P);
```

causes PL/I to declare `P` contextually with the `POINTER` attribute.

- If the identifier appears as the argument of the `BASED` attribute in the `DECLARE` statement for some other identifier. For example, if `P` is not explicitly declared, the statement

```
DECLARE S FIXED BASED(P);
```

causes PL/I to declare P contextually with the POINTER attribute.

The AREA Attribute: Certain contexts cause an undeclared identifier to be declared contextually with the AREA attribute. These contexts are as follows:

- IN(identifier) option in the ALLOCATE and FREE statements
- OFFSET(identifier) attribute of the DECLARE statement

Other Remarks: The above paragraphs contain a complete list of those contexts that, according to ANSI rules, cause PL/I to make a contextual declaration of an identifier that is not explicitly declared. Since Prime supports the ANSI rules, the above is also a complete list of all contextual declarations supported by Prime. To avoid confusion among users who are accustomed to IBM or other older compilers, we mention here a type of contextual declaration that is supported by older compilers but that was discarded by the ANSI committee. In these older compilers, an identifier used as the target of a CALL statement and not explicitly declared would be contextually declared as an EXTERNAL ENTRY. Furthermore, an identifier that is not recognized as a valid built-in function name, but that appears in an expression followed by a parenthesized argument list, is contextually declared as an EXTERNAL ENTRY, if there is no explicit declaration for it.

Therefore, in these older compilers, the statements

```
CALL SBR(A);  
A = FNC(B);
```

would cause SBR and FNC to be contextually declared as having the EXTERNAL ENTRY attributes, provided that there were no explicit declarations for these identifiers. According to the ANSI rules, and therefore according to PRIME rules, the above two statements would be illegal without explicit declarations of SBR and FNC.

There are some final rules concerning contextual declarations:

- If an identifier appears in an explicit declaration, PL/I never makes a contextual declaration for it.
- If an identifier that has not been explicitly declared appears in two or more different contexts that would cause PL/I to make contextual declarations according to the rules in the paragraphs above, then the different contexts must lead to consistent sets of attributes. For example, if the same undeclared identifier appears in both a FILE option and an IN option, then your program is in error, because the FILE attribute is inconsistent with the AREA attribute.

Implicit Declarations

If you use an identifier (not a keyword) in your program, and if the following two conditions hold:

- Your program contains no explicit declaration for the identifier; and
- Nowhere in your program does the identifier appear in any context that qualifies it for a contextual declaration, according to the rules of the preceding section;

then PL/I implicitly declares the identifier. This means that PL/I declares the identifier with the default attributes (usually BINARY FIXED REAL(31,0)). Implicit declarations generate level-1 error warnings at compilation time.

For example, if the variable X appears in your program in the statements

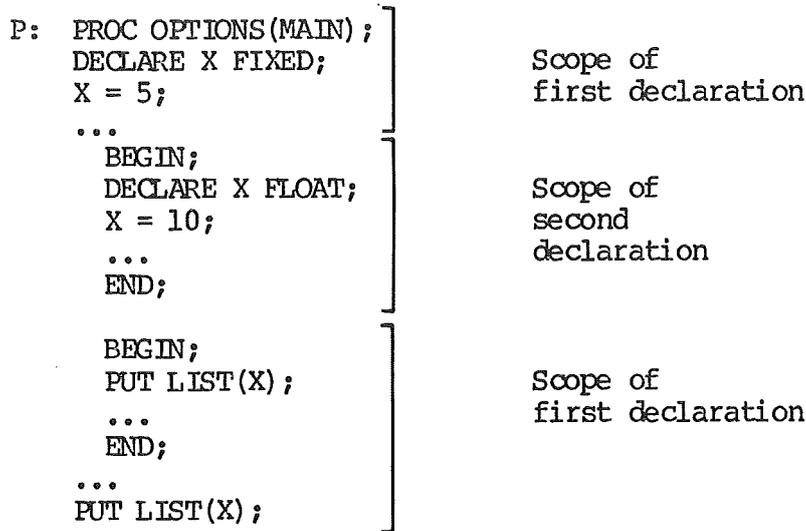
```
X = A + B;
...
PUT LIST(X);
```

and if X appears nowhere else in your program, then PL/I implicitly declares X with the default attributes.

SCOPE OF A DECLARATION

If your program contains an explicit declaration of an identifier, that explicit declaration may or may not apply to your entire program. That portion of your program to which a declaration applies is called the scope of the declaration.

Consider Figure 9-5. This program skeleton contains two explicit declarations of X. The first of these is not inside an internal block, but the second is inside an internal BEGIN block. The scope of the second declaration is the internal BEGIN block, as shown by the bracket lines to the right of the figure. This means that the second declaration applies only within that BEGIN block. The scope of the first declaration of X is all the rest of the program. This means that any reference to X, except within the first BEGIN block, is to the variable declared by the first DECLARE statement.



Scope of Declarations
Figure 9-5

In this program, the statement

```
X = 5;
```

is in the scope of the first declaration, as we have just described, and so the identifier X in this statement refers to the X declared in that declaration. On the other hand, the statement

```
X = 10;
```

is in the scope of the second declaration, and so the X in that statement refers to the X declared by the second DECLARE statement. These two variables, both with the identifier X, are completely different; they have different values and different data types. They are just as different as if they had different identifiers.

The statement

```
PUT LIST(X);
```

appears twice in this program segment, both times within the scope of the first declaration. Therefore, both PUT statements refer to the X that was set to 5 near the beginning of the program. Unless some other statement changes the value of this variable X, each of these PUT statements prints the value 5. In particular, the value of X is not

10, since it is a different X that was set to 10 within the BEGIN block.

The following sections give the precise rules for determining the scope of a declaration.

Block Containment of Explicit Declarations

The scope of an explicit declaration depends upon where the declaration appears in the program as compared to the location of the various internal blocks of the program. In particular, to understand the concept of scope of a declaration, we must understand what it means for a declaration to be inside a block, or contained in a block. In most cases, it is perfectly obvious whether or not an explicit declaration lies within a given block.

The case of the DECLARE statement is completely obvious. A DECLARE statement is contained in a given block if the DECLARE statement lies between the PROCEDURE or BEGIN statement that begins the block and the END statement that ends the block. This means that a DECLARE statement is contained in a given block if it lies within the block in the obvious sense.

Almost as obvious is the case of statement labels for statements other than PROCEDURE, BEGIN, or ENTRY statements. Consider, for example, the following BEGIN block:

```

    BEGIN;
L:  X = 5;
    ...
    END;

```

The explicit declaration for L is contained in the BEGIN block shown.

Another type of explicit declaration is the case of an identifier in a parenthesized list following the first keyword in either a PROCEDURE or ENTRY statement. Such an identifier is thereby explicitly declared to be a PARAMETER. This explicit declaration is contained in the PROCEDURE block for the PROCEDURE or ENTRY statement.

The only remaining case of an explicit declaration is the one that is least obvious, the case of a statement label for a BEGIN, PROCEDURE, or ENTRY statement. In all three cases, the explicit declaration determined by the statement label is not contained in the block defined by the PROCEDURE or BEGIN statement, or by the PROCEDURE statement corresponding to the ENTRY statement. However, it is contained in the next outer block.

Although it seems confusing, this is indeed true for the label on an ENTRY statement. Even though such a label appears clearly to be

defined inside the procedure in which the ENTRY statement lies, PL/I considers that declaration to be outside the procedure.

In the illustration below, SUBR is an internal procedure with a secondary entry point, SUBR2. The box drawn in the example shows which explicit declarations are contained in the PROCEDURE block. The explicit declarations of A, B, C, and L are contained in the PROCEDURE block, but the explicit declarations of SUBR and SUBR2 are not contained in that block.

```

...
SUBR:  PROCEDURE(A);
       DCL A FLOAT, B FIXED;
L:     B = 10;
SUBR2: ENTRY(A,C);
       ...
       END SUBR;
...

```

Immediate Containment

Consider the next example. The procedure SUBB contains the declarations of both A and B. However, the procedure does not immediately contain the declaration for B, since that declaration is inside an internal block. The PROCEDURE block immediately contains the declaration for A, and the BEGIN block immediately contains the declaration for B.

```

SUBB:  PROCEDURE;
       DECLARE A FIXED;
       ...
       BEGIN;
       DECLARE B FIXED;
       ...
       END;
END SUBB;

```

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

A given block is said to contain immediately a given explicit declaration if the block contains the declaration, but no inner block also contains the declaration. Note that several blocks can contain a given declaration, but only one block can contain a declaration immediately.

Scope of an Explicit Declaration

That portion of your program to which an explicit declaration applies is called the scope of the explicit declaration.

To get the scope of an explicit declaration,

- Start with the block that immediately contains the explicit declaration. This includes all statements in this block, including statements that are inside internal blocks within the block.
- Look for other explicit declarations of the same identifier in internal blocks. Cross out any internal blocks that immediately contain those explicit declarations.

Figure 9-6 illustrates the results of applying these rules. This program skeleton contains a number of blocks, labelled P, S, U, V, W, and X, respectively. Three declarations of B are shown. For the first of these three declarations, the one following the PROCEDURE statement for procedure S, the scope is shown by the brackets to the right of the program skeleton. We obtained this result by starting with procedure S, the block that immediately contains the first declaration, and then crossing out blocks V and W, since these blocks immediately contain explicit declarations of B. The result is the scope of the first declaration.

```

P:  PROC OPTIONS(MAIN);
    ...
S:  PROC;
    DECLARE B FIXED;
    ...
U:  BEGIN;
    ...
V:  BEGIN;
    DCL B FLOAT;
    ...
    END V;
    ...
    END U;
    ...
W:  BEGIN;
    DECLARE B;
    ...
    END W;
    ...
    END S;
    ...
X:  BEGIN;
    ...
    END X;
    END P;

```

Scope of the First Explicit Declaration of B
Figure 9-6

Scope of an Implicit or Contextual Declaration

PL/I considers an implicit or contextual declaration to be immediately contained in the external procedure being compiled. With this information, we can apply the general rules of the last section to get the following rules for the scope of an implicit or contextual declaration:

- Start with the entire external procedure.
- Cross out all internal blocks that immediately contain an explicit declaration for the same identifier.

The result is the scope of the implicit or contextual declaration.

Figure 9-7 illustrates these rules. This figure shows an entire program consisting of an external procedure, P, and three internal BEGIN blocks, U, V, and W. There is an explicit declaration of the identifier B within the BEGIN block V, and so the scope of this explicit declaration is just the block V. There are other references to the identifier B outside of the BEGIN block V, and there is no explicit declaration for these references. Therefore PL/I creates an

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

implicit declaration for B. This implicit declaration is immediately contained in the external procedure P. To get the scope of this implicit declaration, we apply the above rules. We start with the external procedure P, and we cross out the BEGIN block V that immediately contains an explicit declaration of the same identifier, B. The result is the scope of the implicit declaration for B, and it is indicated by the brackets to the right of the program in the figure. If this program is executed, the PUT statement prints the value 5, which is the value to which B was set inside the BEGIN block U.

```
P: PROC OPTIONS(MAIN);  
    U: BEGIN;  
        B = 5;  
    END U;  
    V: BEGIN;  
        DECLARE B;  
        B = 10;  
    END V;  
    W: BEGIN;  
        PUT LIST(B);  
    END W;  
END P;
```

Scope of an Implicit Declaration of B
Figure 9-7

RESOLVING REFERENCES

Whenever any statement of your PL/I program references an identifier that is not a keyword, the PL/I compiler must match that reference up with some declaration, explicit, implicit, or contextual. This process is called resolving the reference.

Multiple Declarations

As a general rule, if a program contains two declarations for the same identifier, they must be immediately contained in different blocks of your program. As a result, the two declarations have different scopes, and, in fact, the scopes never overlap.

There are some exceptions to this general rule. Some of these exceptions have been illustrated in earlier sections; all of them are summarized here. The precise rule for two or more explicit declarations of the same identifier is as follows: two or more explicit declarations of the same identifier may not be immediately contained in the same block of your program, unless each pair of such explicit declarations meets one of the following conditions:

- At least one of the two explicit declarations in the pair is for a member of a structure. To put it another way, they must not both be level-1 declarations of the same identifier.
- The two explicit declarations in the pair are for parameters in the parameter lists of different PROCEDURE or ENTRY statements immediately contained in the same PROCEDURE block.
- One of the explicit declarations is for a parameter in the parameter list of a PROCEDURE or ENTRY statement, and the other is a separate declaration of the same identifier in a DECLARE statement, the purpose of which is to specify the data type and aggregate type of the parameter.
- Both explicit declarations are subscripted labels for LABEL CONSTANT arrays. In this case, it is required that the subscripts be constant, that both references have the same number of subscripts, and that the values of the subscripts be different.

All of the above exceptions, except the first, have been illustrated earlier in this chapter. The next section illustrates the first of these exceptions.

Structure References

Suppose your program contains the following code and you wish to assign the value 5 to the element A.B.C. Consider the four different assignment statements at the end of the code. The reference to A.B.C in the first statement of this example is called a fully qualified reference. The others are called partially qualified references, since one or more intermediate structure names are missing. If the program contains no other declarations for the identifiers A or B or C, these four assignment statements are all legal, and they are all equivalent.

PROGRAM BLOCKS, DECLARATIONS, AND SCOPE RULES

```
DECLARE 1 A,  
        2 B,  
        3 C,  
        3 D,  
        2 E;  
A.B.C = 5;  
B.C   = 5;  
A.C   = 5;  
C     = 5;
```

If there are other declarations of C in the program, not all of the references in this example are legal. In such a case, the statement C = 5 is certainly ambiguous, and is therefore invalid. The statement A.B.C = 5, which contains the fully qualified reference, is always valid. Whether the other two statements are valid depends on whether there are any other declarations in the program that make these statements ambiguous.

Interleaved Subscripts

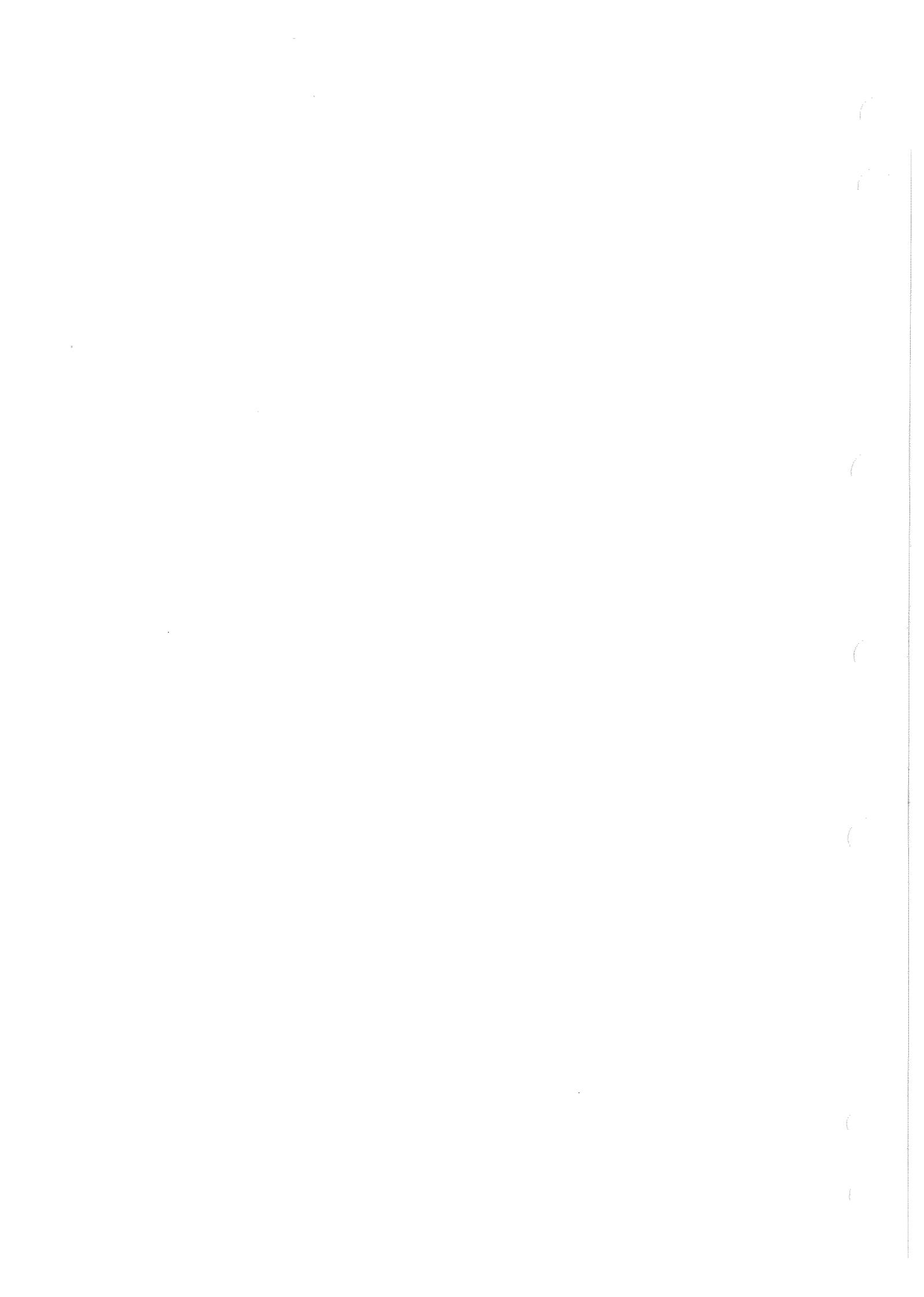
Consider this declaration:

```
DECLARE 1 S(10),  
        2 A(5),  
        3 B,  
        3 C(20),  
        2 D;
```

A reference to S(I).A(3).C(K) is said to have interleaved subscripts, since the subscript lists come between the various identifier qualifier levels.

PL/I permits such a reference to be written with all subscripts to the right. For the above example, this would be S.A.C(I,3,K), and, in fact, PL/I considers these two references to be completely equivalent.

The rules for resolving structure references would then apply to S.A.C, with the subscripts ignored for the purpose of reference resolution.



10

Flow of Control

This chapter examines in detail the order in which statements are executed. Normally, when PL/I executes a program, it executes the statements of the program sequentially; that is, it executes one statement of the program, and then executes the statement immediately following. However, certain types of statements, such as those that define program loops or certain types of conditions such as program errors, can alter the sequential execution of a program.

The last part of Chapter 10 presents five statements that direct the compiler to copy text, replace characters, skip to a new page, or suppress and restart the printing of a source listing. The statements do not themselves cause any object code to be generated.

THE IF STATEMENT

Introductory material on the IF statement is in Chapter 4. The format of the IF statement is

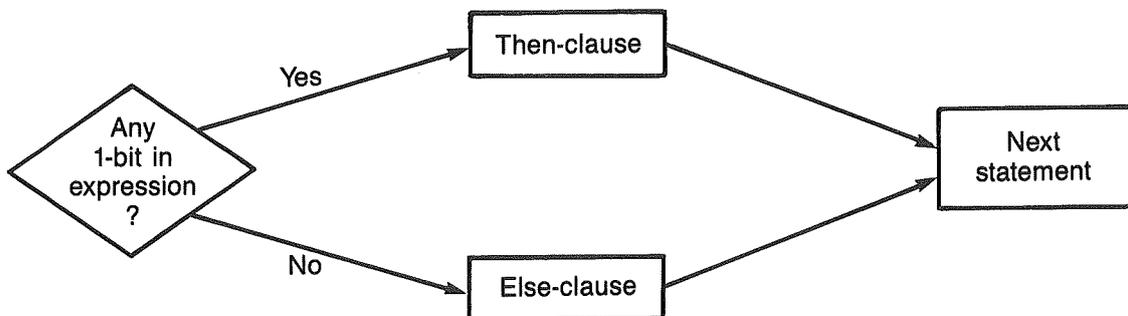
```
IF expression THEN then-clause;  
[ELSE else-clause;]
```

The expression is any PL/I expression. The then-clause and the else-clause are each one of the following:

- A single executable statement of one of the following types: ALLOCATE, =, CALL, CLOSE, DELETE, FREE, GET, GOTO, LOCATE, NULL, OPEN, PUT, READ, RETURN, REVERT, REWRITE, SIGNAL, STOP, or WRITE.
- Another IF statement, with its own subclauses.
- An ON statement, possibly with its own ON-unit (discussed in Chapter 13).
- A group of statements beginning with a DO statement and ending with an END statement (discussed later in this chapter).
- A block of statements beginning with a BEGIN statement and ending with an END statement. For performance reasons, a block is not recommended unless you need the full power of PL/I's block mechanism.

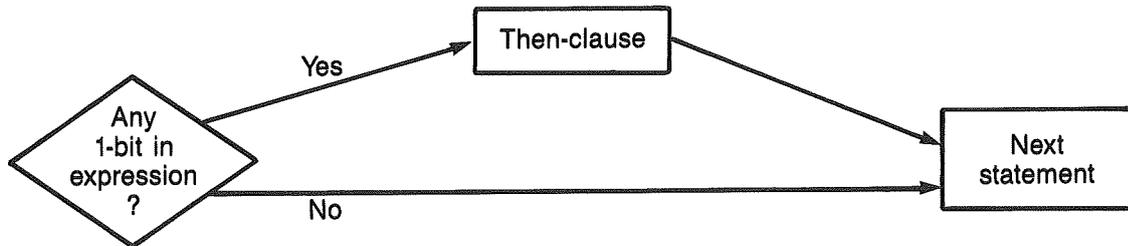
PL/I executes the IF statement by first determining whether the expression is true or false. It does this by evaluating the expression and converting it to the BIT data type according to the rules given in Chapter 6. If the resulting BIT string contains any 1-bit, PL/I considers the expression to be true; if the BIT string is null, or if it contains only 0-bits, PL/I considers the expression to be false.

Figure 10-1 is a flow chart showing the operation of the IF statement. PL/I executes either the then-clause or the else-clause, depending upon whether or not there is a 1-bit in the result obtained by evaluating the expression and converting it to BIT. After executing either one clause or the other, control passes to the next statement. Of course, this does not happen if the clause that is executed specifies a change in flow of control, such as by means of a GOTO statement.



Flow of the IF Statement
Figure 10-1

The IF statement format shown above uses the brackets ([]) to indicate that you need not specify the ELSE option. If you use an IF statement with no ELSE option, the execution is as shown in Figure 10-2.



IF With No ELSE
Figure 10-2

Nested IF Statements

If either the then-clause or the else-clause of an IF statement is another IF statement, we have a case of nested IF statements. This is illustrated in the following example. The nested IF statements are combined to test the various possibilities shown in the chart below.

```

IF X > 0 THEN IF X <= 10
    THEN A = 1;
    ELSE A = 2;
ELSE IF X = 0
    THEN A = 3;
    ELSE A = 4;
  
```

Various values of X have the following results:

<u>Tested Value of X</u>	<u>Value Assigned to A</u>
> 0 but <= 10	1
> 10	2
0	3
< 0	4

The danger in using nested IF statements is that it is very easy to make a mistake in matching the ELSE options with the appropriate IF statements. To understand the problem, look at the next example.

There is a missing ELSE clause in that example. To which of the two IF statements does the single ELSE keyword belong?

```
IF X > 0 THEN IF X <= 10
           THEN A = 1;
           ELSE A = 2;
```

PL/I always matches each ELSE with the nearest unmatched IF. In the example, this means that PL/I matches the ELSE with the second IF, which may or may not have been the programmer's intention. For the ELSE clause to match with the first IF, there must be a dummy ELSE clause, as follows. In this example, the line ELSE specifies an ELSE clause consisting of a null statement. This ELSE matches the second IF, so that the second ELSE can match the first IF.

```
IF X > 0 THEN IF X <= 10
           THEN A = 1;
           ELSE;
           ELSE A = 2;
```

THE DO STATEMENT

Introductory material on the DO statement is in Chapter 4.

The DO statement has two purposes:

- To define a statement group. The collection of statements between the DO statement and its corresponding END statement form a group that, for several purposes, can be treated as a single unit. We have already seen this use with the IF statement in the THEN and ELSE clauses.
- To provide looping, by means of repetitive execution of such a group of statements.

DO Statement with No Options

The simplest form of the DO statement is

```
DO;
statements
END;
```

In this format, where the DO statement has no options, we have merely defined a group of statements. This is the most common format used in

the subclauses of the IF statement. No repetition is provided by this format.

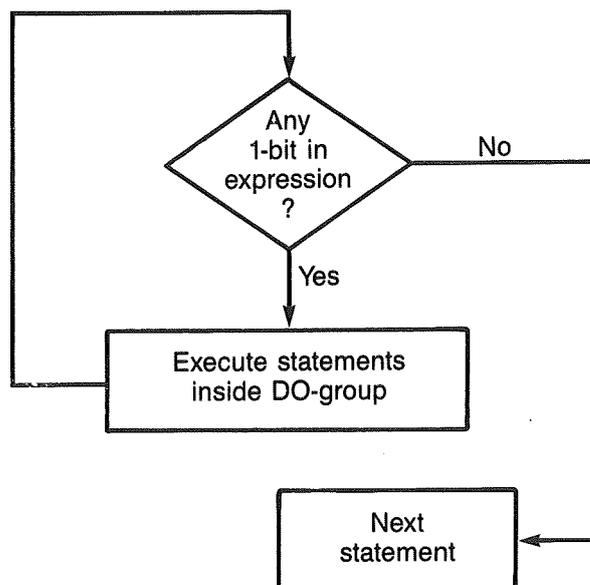
DO WHILE

The simplest form of iterative DO statement has only a WHILE option. The format is

```
DO WHILE(expression);
statements
END;
```

Before each iteration of the loop, PL/I must determine whether the expression is true or false. It does this by evaluating the expression, converting the result to the BIT data type and then considering the expression to be true if there is a 1-bit in the resulting BIT string, false otherwise. As long as the expression remains true, PL/I continues to re-execute the statements in the group.

The operation of the DO loop shown in the format above is flowcharted in Figure 10-3. Notice that PL/I evaluates the expression before the first iteration of the loop, and then re-evaluates it after each subsequent iteration.



Operation of DO WHILE
Figure 10-3

One consequence of these rules is that a zero trip DO loop is possible. This is a loop in which no iterations are executed. An illustration follows. If the first GET statement inputs a negative value of A, the expression $A \geq 0$ is false immediately. As a result, there are no iterations of the loop, and PL/I immediately transfers control to the statement following the END statement.

```
GET LIST(A);
DO WHILE(A >= 0);
  PUT LIST(A);
  GET LIST(A);
END;
```

Another consequence of the above rules is that execution of the loop does not terminate in the middle of an iteration, even if the expression becomes untrue in the middle of an iteration. Consider the example below. If, after several iterations, the GET statement inputs a negative value, the PUT statement still executes. The loop does not terminate until the test of the expression is made at the completion of the iteration.

```
A = 1;
DO WHILE(A >= 0);
  GET LIST(A);
  PUT LIST(A);
END;
```

DO with Numeric Index Variable

The DO statement in FORTRAN, the FOR statement in BASIC, and the PERFORM statement in COBOL give the user the capability of looping under the control of an index variable.

In PL/I, the simplest format of DO with an index is

```
DO index = initial-expr [BY by-expr] [TO to-expr];
statements
END;
```

In this format, the index is the DO loop variable. PL/I begins execution of such a group by initializing the value of the index variable to the value of the initial-expr.

The BY clause and the TO clause are optional. If both are specified, they may appear in either order. The rules for execution of this format DO statement depend upon whether the BY option is specified and

whether the TO option is specified. The rules are given in the remainder of this section.

The simplest case,

```
DO index = initial-expr;
```

has no BY option or TO option. PL/I initializes the index variable to the value of the initial-expr, and then executes the statements in the group once. There are no multiple iterations. For example, the DO group

```
DO X = 50;
PUT LIST(X);
END;
```

contains a PUT statement that is executed only once, with the value of X equal to 50.

Next, let us consider the case where you specify only the BY option:

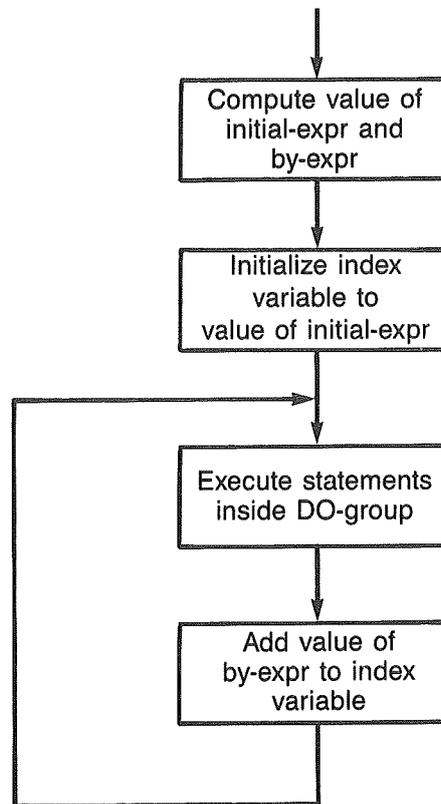
```
DO index = initial-expr BY by-expr;
```

If PL/I executes a loop beginning with this DO statement, the result is usually an infinite loop. PL/I initializes the index variable to the value of the initial-expr, and then executes the statements inside the group. After all the statements have been executed, PL/I adds the value of the by-expr to the index variable, and then executes the statements in the group again. The process of modifying the index variable and executing the statements in the group continues indefinitely. Such a DO loop cannot terminate normally. It continues looping until either the program executes a GOTO statement that transfers out of the loop, or until the program is stopped by some external means.

Figure 10-4 shows, by means of a flowchart, how PL/I executes a DO loop of this type. For example, the loop

```
DO K = 1 BY 1;
PUT LIST(K);
END;
```

prints the values 1, 2, 3, 4, and so forth, until something external terminates the program.



Indexed DO Loop
Figure 10-4

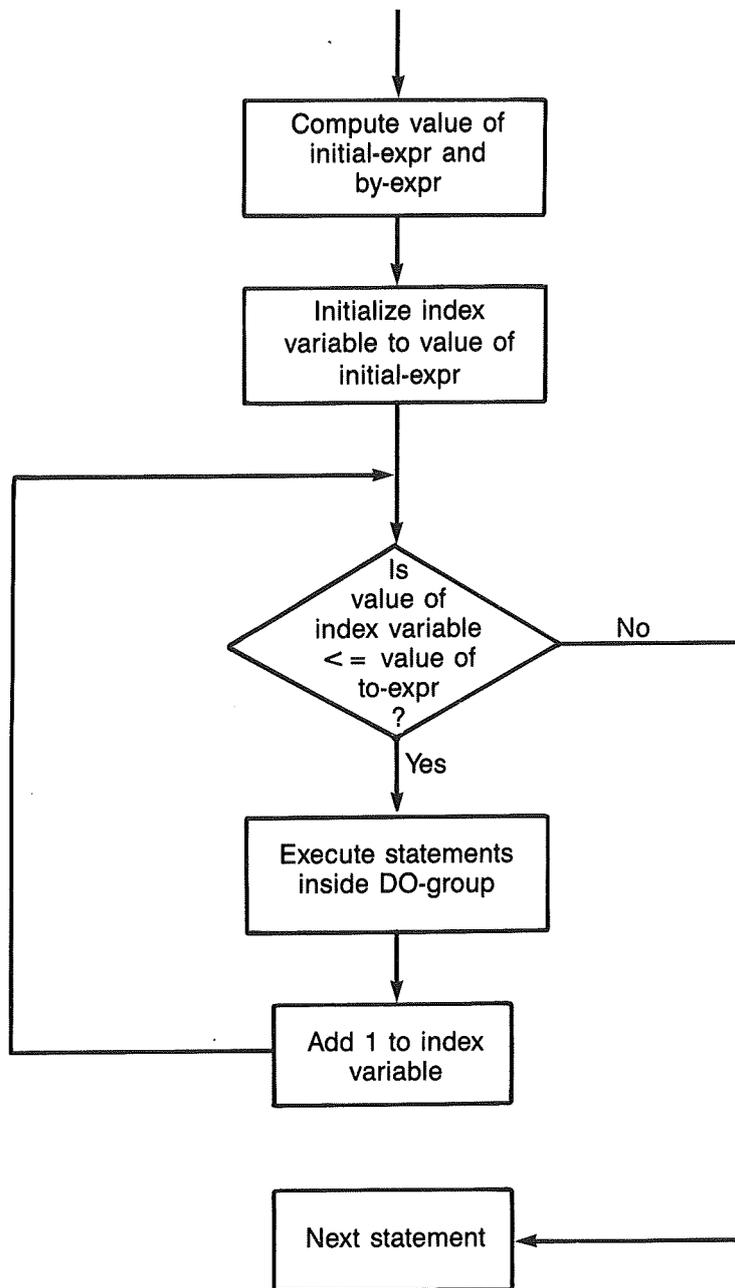
Next, let us consider the case where only the TO option is specified:

```
DO index = initial-expr TO to-expr;
```

The TO option specifies an ending value for the index variable, to terminate the loop normally. PL/I executes the loop beginning with this type of DO statement by initializing the index variable to the value of the initial-expr for the first iteration of the loop, and then incrementing the index variable by 1 for each new iteration, stopping when the value of the to-expr is reached.

Figure 10-5 is a flowchart that shows how PL/I executes a DO group that begins with this type of DO statement. For example, the following program segment prints the values 1, 2, 3, 4, and 5:

```
DO K = 1 TO 5;  
  PUT LIST(K);  
END;
```



DO Without BY
Figure 10-5

As you can see from the flowchart, PL/I decides whether to execute each new iteration of the statements in the DO group by testing whether the

index variable is less than or equal to the value of the to-expr. This means that the loop terminates even if the index variable comes to exceed the value of the to-expr, without ever actually equalling it. For example, the next program segment prints the values 2.5, 3.5, 4.5, and 5.5. The variable X will never actually equal 6, because the loop terminates after 5.5 has been printed. Although the last printed value is 5.5, the variable X has a value of 6.5 as the loop is terminated.

```

DECLARE X FIXED DECIMAL(7,1);
  DO X = 2.5 TO 6;
  PUT LIST(X);
END;

```

The initial-expr and the to-expr may be arbitrary PL/I expressions. PL/I evaluates each of them only once, when the DO loop is entered. To understand the significance of this, consider the following example. The loop in that example prints the values 1, 2, 3, 4, and 5. Note that PL/I evaluates the to-expr only once to get the value 5, and does not re-evaluate the expression after subsequent loop iterations. Therefore, the change in the value of M to 2 during execution of the statements in the group has no effect on the value of the to-expr.

```

M = 5;
  DO K = 1 TO M;
  PUT LIST(K);
  M = 2;
END;

```

It is apparent from the flowchart in Figure 10-5 that a zero-trip DO loop is possible. Consider the program segment below. The way in which PL/I executes the loop depends upon the value assigned to A as a result of the GET statement. If A equals 7, the loop prints the values 5, 6, and 7. But if A equals 2, the loop sets K to the value 5 but does not execute the PUT statement at all, resulting in a zero-trip loop.

```

GET LIST(A);
  DO K = 5 TO A;
  PUT LIST(K);
END;

```

As the flowchart shows, the index variable always has a well-defined value when the loop terminates. Upon normal termination of the loop, the index variable always exceeds the value of the to-expr. This means that, except in the zero-trip case, the final value of the index variable is one greater than the value it had during the last iteration of the loop. This is illustrated by the next example, where the loop

terminates with a value of K equal to 51. Therefore, the final PUT statement prints the value 51.

```
DO K = 1 TO 50;  
  ...  
  END;  
PUT LIST(K);
```

The final format uses both the TO and BY options:

```
DO index = initial-expr BY by-expr TO to-expr;
```

or

```
DO index = initial-expr TO to-expr BY by-expr;
```

This is the most complicated case, but it is similar to the preceding case (TO option but no BY option), except that, at the end of each iteration of the loop, PL/I increments the index variable by the value of the by-expr, rather than by 1.

For example, the next loop prints the values 1, 4, 7, 10, and 13. At normal termination of the loop, the value of K is 16.

```
DO K = 1 BY 3 TO 14;  
  PUT LIST(K);  
  END;
```

There is an important special case, the case where the value of the by-expr is negative. For example, the loop

```
DO K = 5 TO 1 BY -1;  
  PUT LIST(K);  
  END;
```

prints the values 5, 4, 3, 2, and 1. When the loop terminates normally, the value of K is 0.

The rules for when the expression in the BY option is negative are similar to those for when no BY option is used, except that

- PL/I decreases, rather than increases, the value of the index variable at the end of each iteration; and
- The test for loop termination is whether the index variable is less than, rather than greater than, the value of the to-expr.

WHILE Option With an Index Variable

Consider the following program segment:

```
SUM = 0;
  DO K = 1 TO 15 WHILE(SUM <= 20);
    SUM = SUM + K;
  END;
  PUT LIST(K, SUM);
```

The loop in this example contains two different conditions for termination:

- The value of K exceeds 15, as specified by the TO option.
- The value of SUM exceeds 20, as specified by the WHILE option.

PL/I terminates the loop as soon as either of these conditions occurs, whichever comes first.

In the case of the program segment just above, the following happens: during the sixth iteration, when K equals 6, the statement

```
SUM = SUM + K;
```

sets the value of SUM to 21. As a result of the specification in the DO statement, PL/I increments the value of K to 7. The loop terminates at this point, not because of the TO option, but because of the WHILE option, since the value of SUM exceeds 20. At this normal termination, the value of K is 7 and the value of SUM is 21. These are the values that are printed by the final PUT statement in the example.

To clarify the precise rules, suppose your DO statement has the following format:

```
DO index = initial-expr TO to-expr WHILE(while-expr);
```

Figure 10-5 was a flowchart showing how PL/I handles this DO statement format when there is no WHILE clause. Figure 10-6 is a modification of that flowchart to show what happens when there is a WHILE clause. Notice in particular that PL/I evaluates the initial-expr and the to-expr only once, when the loop begins. (This is also true of the by-expr when you specify a BY option.) On the other hand, the while-expr is evaluated for each iteration of the loop.

PL/I Reference Guide

Similar rules apply to other cases when you add a WHILE clause to an index variable specification. For the format

```
DO index = initial-expr WHILE(while-expr);
```

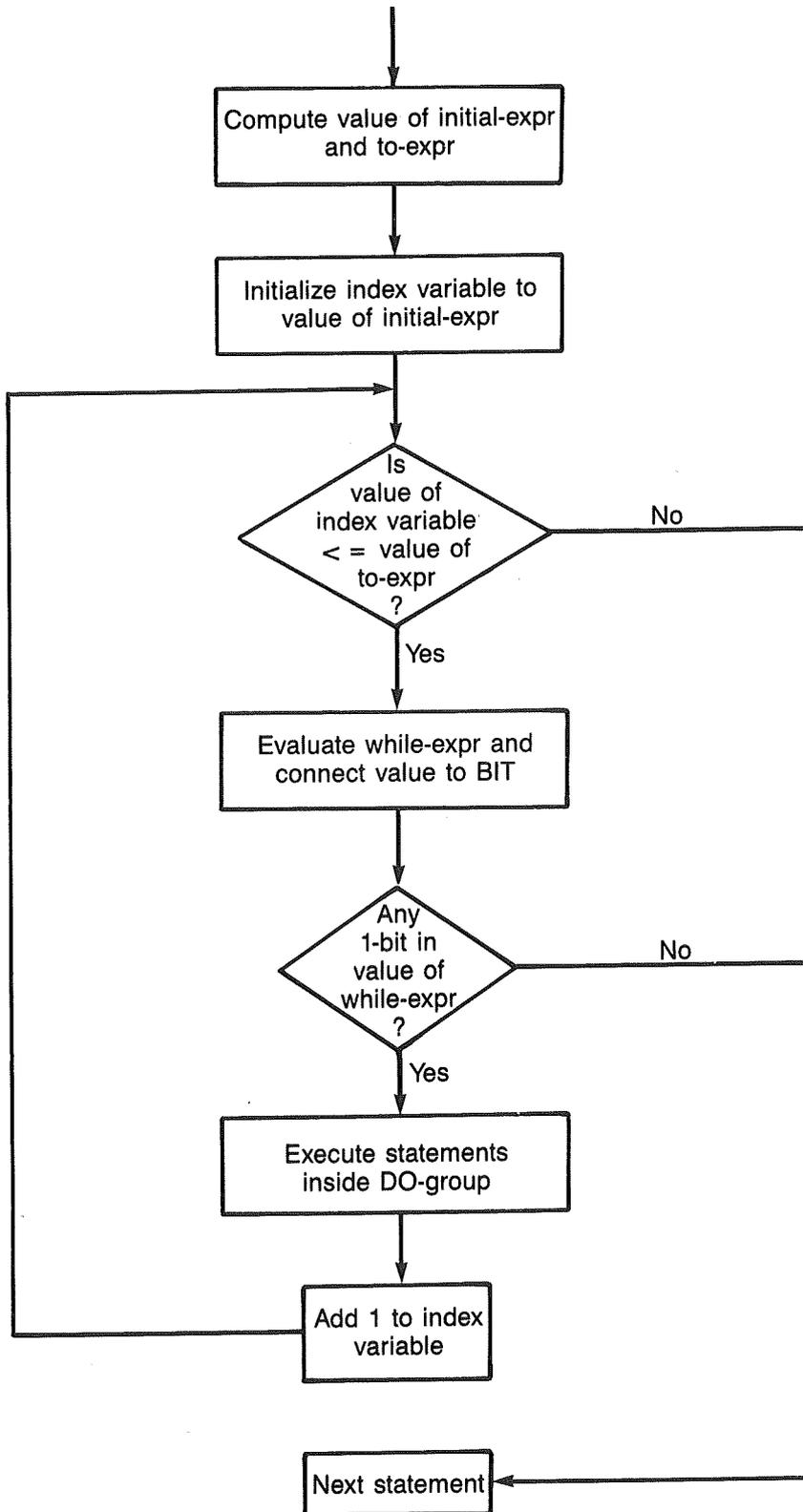
PL/I executes the statements inside the group at most one time. If the while-expr is true, PL/I executes the statements once. There is no execution at all if that expression is false.

As we have already seen, the case where there is a BY clause but no TO clause leads to an infinite loop. However, this is no longer the case when you add a WHILE clause. A group beginning with a statement in the format

```
DO index = initial-expr BY by-expr  
  WHILE(while-expr);
```

repeats indefinitely as long as the while-expr remains true. When it becomes false the loop terminates normally. The next illustration contains a DO statement of this type. This loop terminates normally after six iterations, because the value of SUM exceeds 20. The PUT statement in that example prints the values 7 and 21.

```
SUM = 0;  
  DO K = 1 BY 1 WHILE(SUM <= 20);  
    SUM = SUM + K;  
  END;  
PUT LIST(K, SUM);
```



DO WHILE With an Index Variable
Figure 10-6

The Complete Do Statement

The full format of the DO statement is

1.
$$\text{DO} \left[\begin{array}{l} \text{WHILE}(\text{expression}) \text{ [UNTIL}(\text{expression})] \\ \text{UNTIL}(\text{expression}) \text{ [WHILE}(\text{expression})] \end{array} \right];$$
2.
$$\text{DO index} = \text{expr} \left[\begin{array}{l} \text{TO expr [BY expr]} \\ \text{BY expr [TO expr]} \\ \text{REPEAT expr} \end{array} \right] \text{ [WHILE}(\text{expr})] \text{ [UNTIL}(\text{expr})];$$

The REPEAT clause is explained below.

The UNTIL clause is a Prime extension and is similar to WHILE. The expression following UNTIL is a logical expression. It allows the programmer to assure that the DO statement does not loop indefinitely, by establishing a condition for termination. The expression after UNTIL is evaluated after each execution of DO and, when it is true, control passes to the statement following DO. An UNTIL clause thus ensures that the DO group is executed at least once. An example is

```
DO UNTIL(K > 100);
K = K + 1;
END;
```

Index Variable with REPEAT Option

The REPEAT option is a more general form of the BY option. While the BY option allows you to add a given value to the index variable after each iteration of the loop, the REPEAT option lets you modify the index variable in any way you want. For example, consider

```
DO K = 1 REPEAT 2 * K;
PUT LIST(K);
END;
```

This is an infinite loop. The initial value of K is 1, and after each iteration, PL/I computes the value of the expression 2 * K and assigns that as the new value of K. This means that, for each repetition, the value of K is twice the value of K from the preceding iteration. Therefore, this loop prints the values 1, 2, 4, 8, 16, 32, 64, ..., looping indefinitely until the program halts for an external reason.

PL/I does not permit you to use the TO option with the REPEAT option. Therefore, you have to use the WHILE option to prevent an infinite loop. Consider this example:

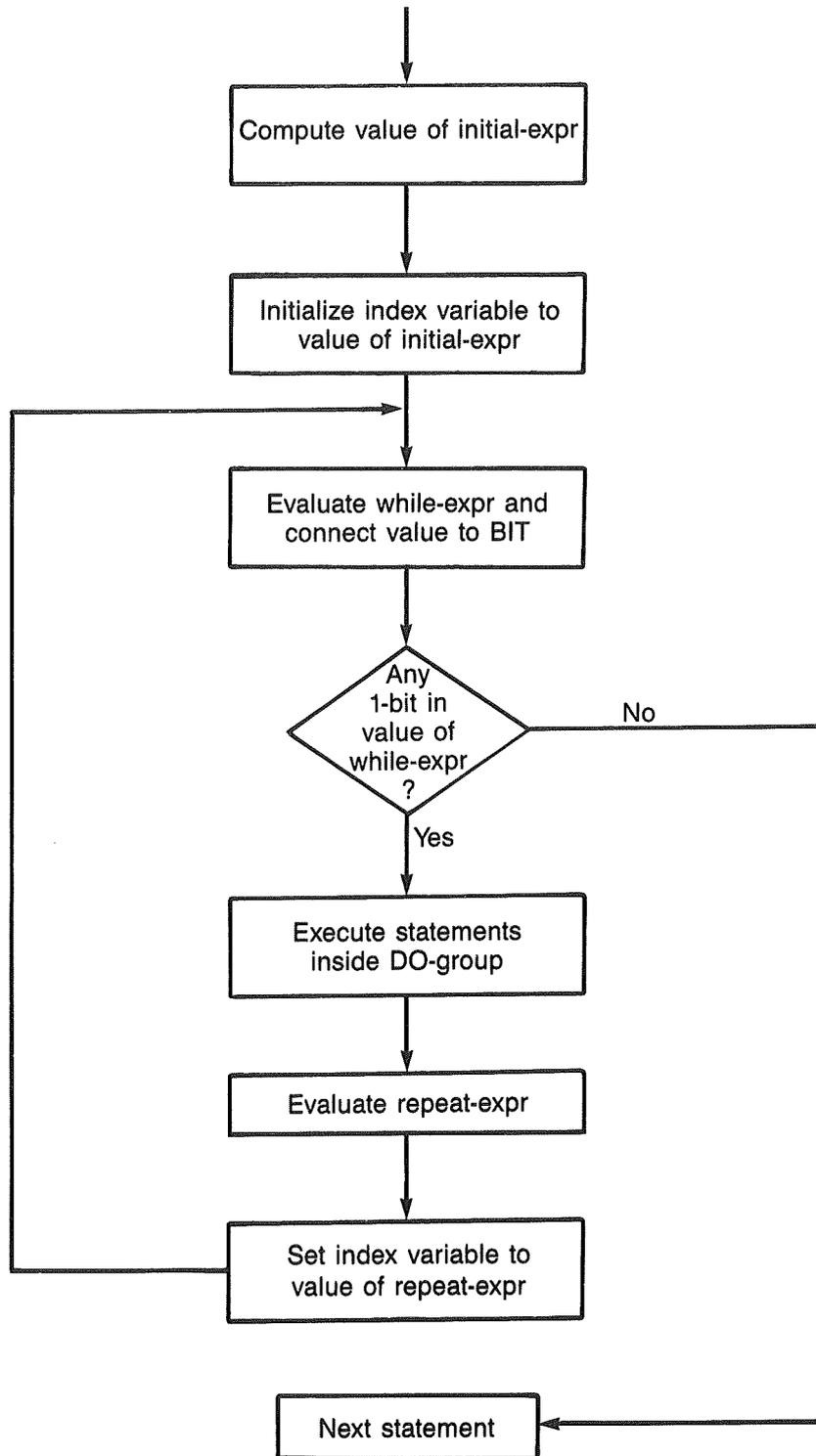
```
DO K = 1 REPEAT 2 * K WHILE(K <= 32);  
  PUT LIST(K);  
END;
```

This loop prints the values 1, 2, 4, 8, 16, and 32. After that, the loop terminates, since the value of K is 64.

To make the rules precise, suppose that the format of the DO statement is as follows:

```
DO index = initial-expr REPEAT repeat-expr  
  WHILE(while-expr);
```

Then PL/I executes the loop as specified in the flowchart in Figure 10-7. Notice that the repeat-expr is like the while-expr, and unlike the by-expr, in that it is evaluated for each iteration of the loop.



The REPEAT Option
Figure 10-7

The importance of the REPEAT option is that, while the BY option can only be used to add a value to the index variable, the REPEAT option can make each new value of the index variable any function of the preceding value. Therefore, we can use something like

```
DO K = 1 REPEAT(F(K)) WHILE(K > 0);
...
END;
```

which actually uses a user-defined function F to compute each new value of K.

Index Variable with Multiple Specifications

We can now summarize all the index variable formats that we have seen so far in this chapter as being in the format

```
DO index = specification;
```

where the specification includes, as we have seen, an initial-expr with optional BY, TO, REPEAT, and WHILE clauses.

We can now expand this format. PL/I permits you to use multiple specifications of this type in a single DO statement, with the specifications separated by commas. For example, in the loop

```
DO K = 1 TO 3, 8 TO 10;
PUT LIST(K);
END;
```

the DO statement has two specifications, 1 TO 3 and 8 TO 10. The loop prints the values 1, 2, 3, 8, 9, and 10.

Another simple example is

```
DO K = 4, 25, 3, -18;
PUT LIST(K);
END;
```

This DO statement has four specifications, 4, 25, 3, and -18. Each of these four specifications is of the type that has only an initial-expr, with no optional BY, TO, REPEAT, and WHILE clauses. The loop prints the values 4, 25, 3, and -18.

To summarize, the format of the DO statement with multiple specifications is

```
DO index = specification, specification, ...;
```

PL/I iterates the loop for each specification in turn. After one specification terminates, PL/I goes on to the next one in the DO statement. Each specification has an initial-expr, with optional TO, BY, REPEAT, and WHILE clauses. When the last specification terminates, the entire loop is considered to have terminated normally.

Nonnumeric Index Variables

In all the examples so far, the index variable has had a numeric data type. PL/I permits you to use an index variable with any data type, including string, pictured, and even noncomputational data types. In such cases, you usually use the REPEAT option to define the iteration rule, and the WHILE option to define the termination condition for the loop. You may not use the TO or BY clause with nonnumeric index variables.

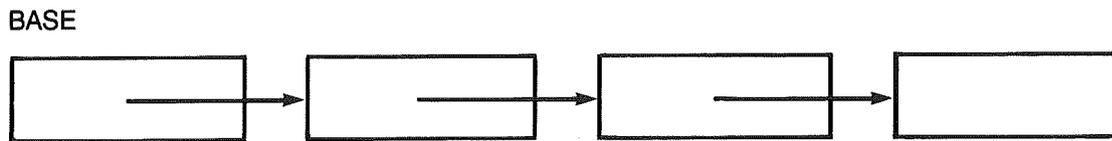
In the next example, C is an index variable with the CHARACTER VARYING data type. During the first iteration of the loop, C has its initial value of 'A'. During subsequent iterations, the value of C is 'AB', 'ABB', 'ABBB', and so forth. The loop prints each of these values, terminating normally after the value 'ABBBBBB' has been printed.

```
DECLARE C CHARACTER(200) VAR;  
DO C = 'A' REPEAT (C || 'B') WHILE(LENGTH(C) <= 7);  
PUT LIST(C);  
END;
```

In list processing applications, it is common to use the REPEAT and WHILE options in a loop to follow along a chain of BASED blocks. For example, suppose your program contains these declarations:

```
DECLARE (P, BASE) POINTER;  
  
DECLARE 1 REC BASED,  
       2 NEXT POINTER,  
       2 VALUE FIXED;
```

These declarations define a linked list that might be pictured as in Figure 10-8.



A Linked List
Figure 10-8

If you wish to print out all the VALUE fields in each of the blocks of the linked list, you can use the following loop. In this loop, the POINTER variable P initially points to the first block in the linked list. For each subsequent iteration, P points to the next block in the list. The loop terminates after the last block in the list has been processed.

```

DO P = BASE REPEAT (P->REC.NEXT) WHILE(P ^= NULL());
PUT LIST(P->REC.VALUE);
END;
  
```

For a data type like ENTRY VARIABLE, you can use multiple specifications in your DO statement if you wish to have a loop index variable with this data type. For example, if EV is a variable with the ENTRY VARIABLE data type, and if E1, E2, and E3 are ENTRY CONSTANT values, the following loop can be used:

```

DO EV = E1, E2, E3;
CALL EV;
END;
  
```

In this DO statement, there are three specifications, each one consisting of an initial-expr.

DO Statement with the IF Statement

Earlier in this chapter, we illustrated the IF statement with THEN and ELSE clauses containing only the simplest forms of DO groups. In fact, any form of the DO statement may be used with the IF statement. For example, the program segment below is legal.

```
IF A >= 0
  THEN DO K = 1 TO 50;
        PUT LIST(K);
        END;

      ELSE DO K = 50 TO 1 BY -1;
        PUT LIST(K);
        END;
```

THE GO TO STATEMENT

Subject to certain restrictions, use the GOTO (or GO TO) statement to transfer control from one part of your program to any other part. The restrictions are that you may not use GOTO to transfer into an inactive group or block. (Active and inactive blocks are presented below in Invocation and Termination of Blocks.)

The format of the GOTO statement is

```
GO TO target;
```

or

```
GOTO target;
```

where target is a LABEL CONSTANT, LABEL VARIABLE, or an expression whose data type is LABEL.

Normal and Abnormal Termination of a Group or a Block

When a GOTO statement transfers control out of a group or a block, the group or block is said to terminate abnormally. Consider the next example. The loop in that example can terminate in two different ways:

- Normally, when the value of K exceeds the value of N;
- Abnormally, when the IF statement executes and the test for SUM being greater than 1000 is successful.

```
GET LIST(N);
SUM = 0;
DO K = 1 TO N;
  SUM = SUM + K;
  IF SUM > 1000 THEN GO TO XL;
END;
```

```
XL: ...
```

Whether the loop actually terminates normally or abnormally depends upon the value of N set as a result of the GET statement.

Some DO loops cannot terminate normally, since the DO statement provides no means for normal loop termination. Some examples of these DO statements are

```
DO K = 1 BY 1;
```

or

```
DO WHILE(2 = 2);
```

or

```
DO X = 2 REPEAT 2 * X;
```

Normal loop termination from a loop beginning with any of these DO statements is impossible, since none of these statements specifies any termination condition that will ever be satisfied. Such a loop will always be an infinite loop, unless the statements inside the group include a GOTO statement that can terminate the loop abnormally.

It is possible for a single GOTO statement to terminate several groups and blocks simultaneously. When the following sample program segment executes, the DO loop calls procedure Q, which calls procedure R. The GOTO statement in procedure R terminates the DO group, as well as the two PROCEDURE block invocations for Q and R.

```
P:  PROC OPTIONS(MAIN);
    ...
    DO K = 1 TO 10;
      CALL Q;
    END;
LB: ...
    ...
Q:  PROCEDURE;
    ...
    CALL R;
    ...
    END Q;

R:  PROCEDURE;
    ...
    GO TO LB;
    END R;
    END P;
```

Whenever a block terminates abnormally, PL/I executes the block epilogue, just as if the block had terminated normally. The block epilogue is described later in this chapter.

The GOTO Statement with LABEL Expressions

Usually the target of a GOTO statement is a LABEL constant. In fact, any expression that has a LABEL value may be used. For example, any of the following can be used:

- LABEL CONSTANT
- Member of a LABEL CONSTANT array
- LABEL VARIABLE
- A reference to a user-defined function that returns a LABEL value

THE LEAVE STATEMENT -- PRIME EXTENSION

This statement provides a means of terminating a group abnormally. The syntax is

```
LEAVE;
```

It causes program execution to be transferred to the statement following the END statement for the current group. In the next example, if ENTRY1 equals 0, the next statement executed is the PUT statement.

```
DO X = 1 TO 100;  
  GET LIST(ENTRY1);  
  IF ENTRY1 = 0  
  THEN LEAVE;  
  TOTAL = TOTAL + ENTRY1;  
END;  
PUT SKIP LIST('END OF RUN');
```

Termination of Multiple Loops with LEAVE

The LEAVE statement in the form just described terminates the innermost DO/END loop in which the DO statement lies. A more general form of the LEAVE statement is

```
LEAVE ident;
```

The "ident" must be the label of a DO statement such that the LEAVE statement lies within the corresponding DO/END group. When this form is used, it is possible for a single LEAVE statement to terminate several DO/END groups simultaneously. Consider the following example:

```
OLUP: DO K = 1 TO 100;
      DO J = K TO 2 * K;
        IF A(K, J) = 0
          THEN CALL RND(A, K);
        ELSE LEAVE OLUP;
      END;
    END;
```

Within this example, the statement

```
LEAVE OLUP;
```

if executed, terminates both DO/END groups, and control will pass to the statement following the second END statement.

THE SELECT STATEMENT -- PRIME EXTENSION

SELECT provides a case selection. A SELECT block has one of two formats:

```
1. SELECT;
   WHEN(if-expression list) statement;
   .
   .
   .
   [OTHERWISE statement;]
   END;
```

```

2. SELECT (value);
   WHEN (value list) statement;
   .
   .
   .
   [OTHERWISE statement;]
   END;

```

In both SELECT formats, the statement is defined to be any simple statement not including DECLARE, END, ENTRY, or PROCEDURE. The statement may include a DO block or a BEGIN block of statements, or be an IF statement. The if-expression list in the first format is either a single expression that evaluates to a BIT(1) result as in an IF statement, or a list of such expressions separated by commas. The value in the second format is any expression that has a scalar value, and value list is either a value or a list of values separated by commas.

A SELECT block is traversed by executing each WHEN clause until a TRUE condition is found. A TRUE condition happens when the if-expression part evaluates to '1'B or a value in the WHEN clause equals a value in the SELECT statement. If a value in the WHEN clause is not of the same data type as the value in the SELECT statement, it is converted to the data type of the latter before the comparison is done. If none of the WHEN clauses is satisfied, if an OTHERWISE clause exists, the OTHERWISE clause is executed; if there is no OTHERWISE clause, ERROR is signalled. After either a WHEN clause or the OTHERWISE clause is executed, control passes to the first executable statement following the SELECT block.

The following example illustrates the second type of SELECT block:

```

GET LIST (INPUT_VALUE);
SELECT (INPUT_VALUE);
WHEN (VALUE_1) CALL UPDATE_RIN;
WHEN (VALUE_2) CALL DELETE_RIN;
OTHERWISE PUT SKIP LIST ('ERROR -- ENTRY IGNORED');
END;

```

If the INPUT_VALUE is equal either to VALUE_1 or to VALUE_2, the appropriate subroutine is called. The OTHERWISE statement catches erroneous input.

PL/I PROGRAM BLOCKS

This section and the next describe in detail how PL/I invokes and terminates program blocks. These sections tie together information presented in other chapters on such subjects as procedures, ON-units, and storage management. They also provide those readers who are interested with the abstract models on which the PL/I block mechanism is based.

Types of Blocks

PL/I recognizes three types of blocks:

- A PROCEDURE block begins with a PROCEDURE statement and ends with an END statement.
- A BEGIN block begins with a BEGIN statement and ends with an END statement.
- An ON-unit is a collection of statements, beginning with a BEGIN statement and ending with an END statement, which an ON statement specifies as the action to be taken when an appropriate error or condition occurs. The ON-unit is considered to be a different kind of block from an ordinary BEGIN block.

Even though the BEGIN block and the ON-unit both begin with a BEGIN statement, PL/I treats these as two quite different types of blocks. When we use the term BEGIN block, we will be referring to one that is not an ON-unit.

Invocation and Termination of Blocks

When your program invokes a block, that block is said to become active. The block remains active until your program terminates it.

The method for invoking a block depends upon the type of block. The methods are as follows:

- Your program invokes a PROCEDURE block by referencing one of the entry points. If the entry point is a subroutine entry point, the reference should be by means of a CALL statement. For a function entry point, your program should reference the function as part of an expression in any type of statement.
- An ON-unit is invoked when the appropriate error or condition occurs. Your program can invoke an ON-unit artificially by means of a SIGNAL statement.

- Your program invokes a BEGIN block by executing the BEGIN statement that begins the block.

It is possible to terminate a block either normally or abnormally. Your program terminates a block abnormally by means of a GOTO statement that transfers control outside of the block.

The way to terminate a block normally depends upon the type of block. The various ways are as follows:

- Your program terminates a PROCEDURE block by executing a RETURN statement. If the PROCEDURE block was invoked as a subroutine, the RETURN statement may not specify an expression; if the PROCEDURE block was referenced as a function, the RETURN statement must specify an expression. If the procedure was invoked as a subroutine, executing the END statement is equivalent to executing a RETURN statement.
- Your program terminates an ON-unit normally by executing the END statement. For some ON conditions, normal termination is illegal. A RETURN statement is illegal inside an ON-unit, unless it appears inside a PROCEDURE block that is itself inside the ON-unit.
- Your program terminates a BEGIN block normally by executing the END statement. If the BEGIN block is inside a PROCEDURE block, executing a RETURN statement terminates both the BEGIN block and the PROCEDURE block. In this case, both terminations are normal.

Prologues, Epilogues, and Storage Management

When your program invokes a block, PL/I executes a prologue before executing any of the statements inside the block. The prologue does the following things:

1. PL/I allocates a dynamic storage area, or DSA. This is described more fully in the next section.
2. If the block is a procedure, PL/I allocates storage for the parameters and establishes pointers to the arguments from the parameters. This is described in Chapter 8. The storage allocated for the parameters is usually part of the DSA.
3. PL/I allocates storage for all AUTOMATIC variables contained immediately in the block. This storage is usually part of the DSA. PL/I then initializes all of the AUTOMATIC variables that are declared with the INITIAL attribute.

When the block terminates, either normally or abnormally, PL/I executes an epilogue for the block. The epilogue frees the DSA, and so frees all AUTOMATIC and parameter storage allocated by the prologue.

Recursion

Recursion occurs when a block that is already active is invoked again. The result is that there can be two or more simultaneous activations of the block.

Chapter 8 illustrates a recursive PROCEDURE block. An ON-unit will also be invoked recursively if the same condition occurs while the ON-unit is already active.

Prime procedures are recursive, whether the RECURSIVE option is selected or not.

The next section explains more fully how recursion works.

STATIC AND DYNAMIC PROGRAM BLOCK STRUCTURE

This section is provided for programmers who need to understand exactly how the PL/I block structure works, especially when recursion is involved. The vast majority of programmers need only the simpler rules given earlier in this chapter.

This text defines the PL/I block structure rules by using an abstract model of how a PL/I program executes with regard to its program block structure. The abstract model is simply a diagram representing both the static and dynamic block structures of the program. The following paragraphs describe how the static diagram is defined when the program is compiled, and how the dynamic diagram changes as the program executes.

It is not the purpose of this section to describe the internal implementation of PL/I. The actual PL/I compiler you are using may or may not use data structures that correspond to the block structure diagrams given in this chapter. These diagrams are only a model of how the PL/I system produces its final results.

Block Structure versus Block Invocations

Consider the program segment shown in Figure 10-9. This program consists of a main procedure called M, with a subprocedure called A.

```
M:  PROCEDURE OPTIONS(MAIN);  
    ...  
    CALL A;  
A:  ...  
    PROCEDURE RECURSIVE;  
    ...  
    END A;  
END M;
```

A Program With Two Blocks
Figure 10-9

From the point of view of the PL/I call system, this program contains two blocks, M and A, with A nested inside M.

From the point of view of the PL/I runtime system, however, the program looks somewhat different. The program begins executing when procedure M is invoked. When M executes the CALL statement, it invokes the subprocedure A. Since A is RECURSIVE, it is possible for A to call itself.

The concept that the program contains two blocks, one nested inside the other, is a static concept in that it does not tell you how the program executes. The block invocation described below is a dynamic concept because the block invocation structure of a program causes it to change constantly as the program executes.

In our example, the program begins executing with the invocation of block M. With the CALL statement there is an invocation of block A. If block A invokes itself recursively, there are several simultaneous invocations of block A. Each time a RETURN is made from A, one of the block invocations disappears, until finally the program returns to M, at which time there is, once again, one block invocation.

Active and Inactive Blocks

During execution of a program, a block is said to be active if an invocation of it exists. A block is inactive when no invocation of it currently exists. During execution of a program, a block may change from inactive to active or from active to inactive many times.

If, during execution of the program, there exists more than one simultaneous invocation of a block, then at that point the block is said to be recursively active.

In Figure 10-9 above, the program begins executing when M becomes active. When the CALL statement is executed, block A becomes active. If A calls itself, block A becomes recursively active. As the

invocation of A terminates, the status of A moves from recursively active to nonrecursively active to inactive.

Inheriting Variables

When your program invokes a new block, PL/I allocates a dynamic storage area (DSA) containing all the AUTOMATIC variables declared within that block.

If the block is an internal block (that is, not an external procedure), and if a statement within the block uses a variable not explicitly declared within the block, then it inherits that variable from the outer block that contains it.

Consider the program in Figure 10-10.

```

M:  PROCEDURE OPTIONS(MAIN);
    DECLARE X;
    X = 5;
    CALL A;
A:  PROCEDURE;
    DECLARE X;
    X = 10;
    CALL B;
    RETURN;
    END A;
B:  PROCEDURE;
    PUT LIST(X);
    END B;
    END M;

```

Invoking and Containing Blocks
Figure 10-10

This program consists of an external procedure M, and two internal blocks called A and B. Notice that there are two declarations of X, one contained immediately in M and the second contained immediately in block A. When this program executes, block M invokes block A, which invokes block B. Inside B, the variable X used is inherited from the containing block, M, rather than the invoking block A. Therefore, the PUT LIST statement prints the value 5.

We use the phrase environmental block invocation to specify the block invocation from which a new block inherits variable values. That is, in Figure 10-10, when block B is invoked, there are two important previous block invocations to consider:

- The invoking block invocation.

This is the block invocation that invoked the new block. In this example, block B is invoked by an invocation of block A.

- The environmental block invocation.

This is the invocation from which the new block inherits the values of variables not declared within the new block. In our example, when block B is invoked, the environmental block invocation is an invocation of block M.

The invoking block invocation can refer to any block in the program, while an environmental block invocation must refer to the block that contains immediately the new block being invoked.

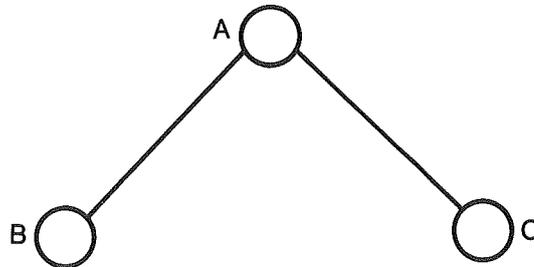
Note the difference between the invoking block invocation and the environmental block invocation. Each of these invocations contributes something to the new block. The invoking block contributes arguments, established on-units, and values of on-condition built-in functions. The environmental block contributes values of variables not explicitly declared within the new block.

The following sections use this abstract model to describe very precisely how the invoking and environmental block invocations contribute these things to the new block.

The Static Block Structure Tree

The abstract model uses two different kinds of diagrams. The first kind of diagram is called the static block structure tree. It represents the block structure of the program as the compiler sees it.

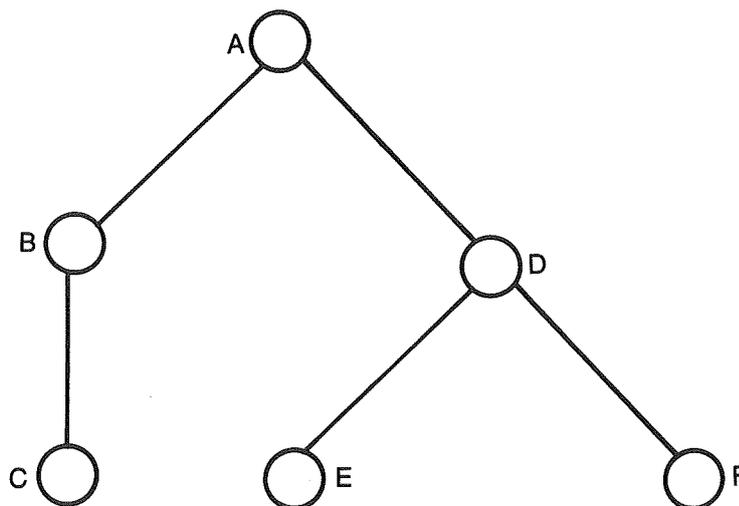
For example, suppose your PL/I program consists of a main procedure called A, which contains two internal procedures called B and C. Figure 10-11 is the static block structure tree for this program. The root node of this tree represents the main procedure, and each branch represents a block contained immediately within the block represented by the root.



Two Blocks Contained Within a Third
Figure 10-11

Let's now consider a more complicated example. Figure 10-12 represents a program whose main procedure is called A. This main procedure contains immediately two internal blocks called B and D. These internal blocks contain their own internal blocks as follows:

- Block B contains immediately an internal block C.
- Block D contains immediately two internal blocks, E and F.

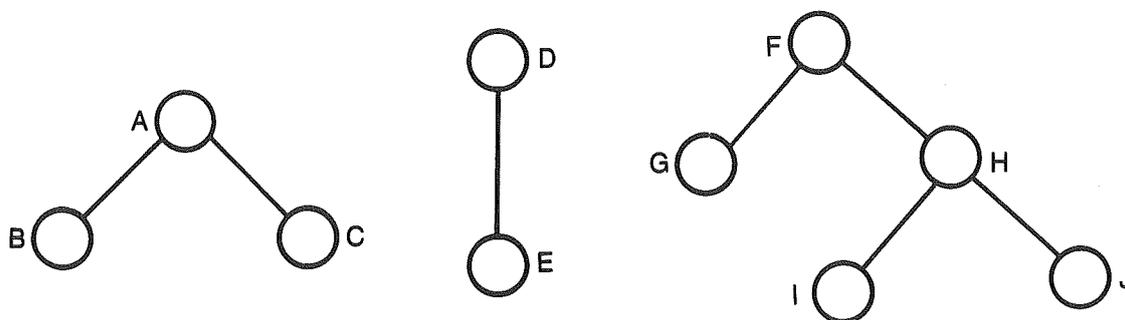


Containment and Immediate Containment
Figure 10-12

Notice that the main procedure A contains five internal blocks, B, C, D, E and F, but only two of these blocks, B and D, are contained immediately.

If your PL/I program contains several external procedures, each external procedure is represented by the root node of a separate tree in the static block structure tree diagram. For example, Figure 10-13 is the static block structure tree diagram for a program containing three external procedures, A, D and F. These external procedures have the following structures:

- The main procedure A contains immediately two internal blocks, B and C.
- The external procedure D contains immediately a single block, E.
- The external procedure F contains immediately two internal blocks, G and H. The internal block H contains immediately two blocks internal to it, I and J.



External Procedures
Figure 10-13

The static block structure tree diagram is the first step in our abstract model of how the PL/I block structure mechanism operates. This part of the model describes the block structure of the program in the way the compiler sees it. We now turn to the block structure of the program as it is seen at runtime.

Dynamic Block Invocation Chain

Consider the program illustrated in Figure 10-14. This program contains a main procedure A, and two internal blocks, B and C.

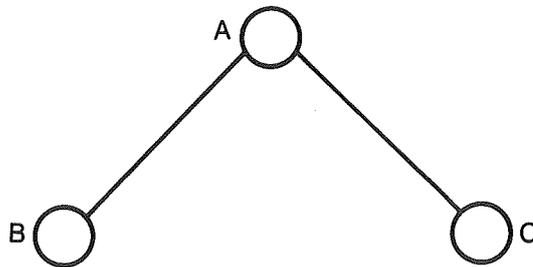
```

A: PROCEDURE OPTIONS(MAIN);
   DECLARE VALUE INIT(0);
B:   BEGIN;
      DECLARE M INIT(2);
      CALL C(M);
      M = M + 1;
      CALL C(M);
      END B;
   PUT LIST(VALUE);
C:   PROCEDURE(K) RECURSIVE;
      DECLARE (L, K) FIXED;
      DO L = 1 TO K - 1;
      CALL C(L);
      END;
      VALUE = VALUE + K;
      RETURN;
      END C;
END A;

```

Block Invocation Chain
Figure 10-14

Figure 10-15 is the static block structure tree diagram for this program. The program begins when PL/I invokes block A. When block A invokes block B, there are two active blocks. If block invocations continue, we may have several active blocks at the same time.



Block Invocation -- Static Diagram
Figure 10-15

We represent the active block invocations in a diagram picturing each block invocation as a circle pointed to by its invoking block.

For example, Figure 10-16 is the diagram of the dynamic block invocation chain for the program illustrated above at the point where

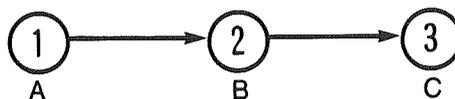
block A has invoked block B. The numbers inside the circles are what we call the block invocation numbers. Each such number simply represents the total number of block invocations that have occurred at the point where this block invocation occurs.



Block Invocation — Dynamic Diagram
Figure 10-16

As Figure 10-16 illustrates, block invocation number 1 is an invocation of block A. It has invoked block invocation number 2, which is an invocation of block B.

If block B now invokes block C, the dynamic block invocation chain diagram is as shown in Figure 10-17. This diagram shows three block invocations, one for each of A, B and C.



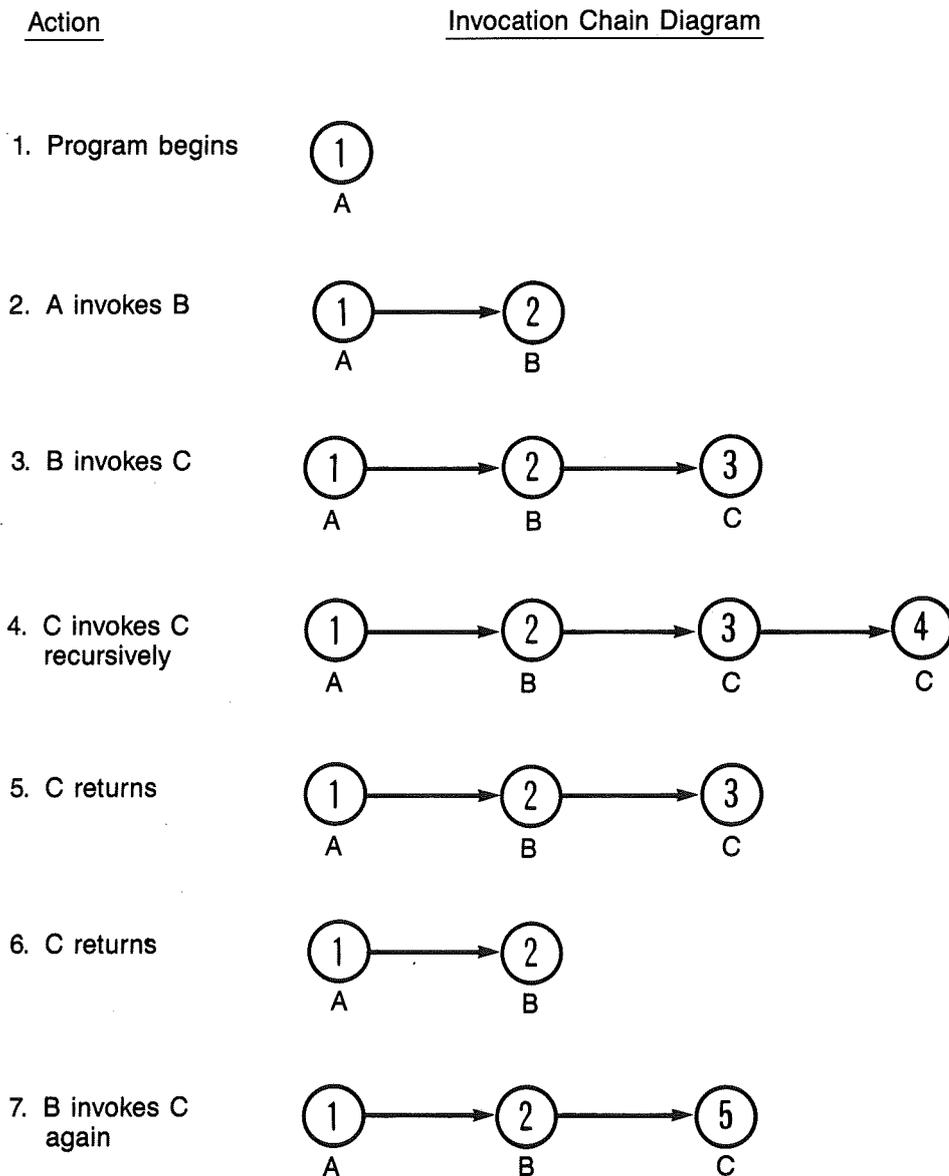
Three Block Invocations
Figure 10-17

If execution of the program continues, the diagram changes. This is illustrated in Figure 10-18. Line 4 of this figure represents what happens after block C invokes itself recursively. There are then four block invocations, and two of these invocations are for block C. When each of these invocations terminates, the result is shown in lines 5 and 6 of the diagram. Notice that the diagram in line 6 is identical to the diagram in line 2.

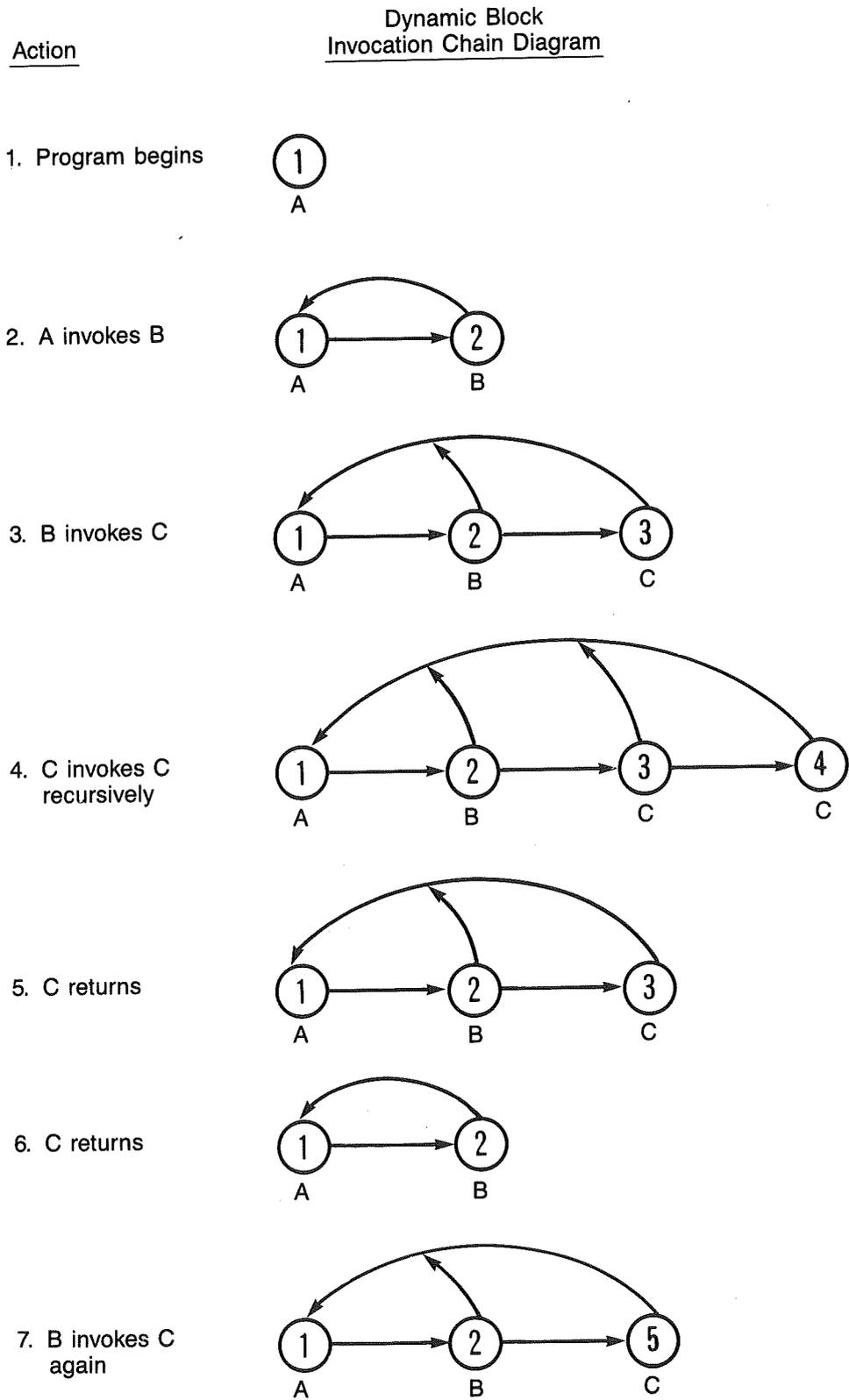
Line 7 shows what happens when block B invokes block C again. Notice that the block invocation number is now 5, since this is the fifth block invocation that has occurred since execution of the program began.

Representing the Environmental Block Invocation

In Figure 10-18, as discussed, the dynamic block invocation chain diagram indicates each block invocation and its invoking block invocation. It is also important that the diagram indicate, for each block invocation, its environmental block invocation.



Changing Block Invocations During Execution
Figure 10-18



Environmental Block Invocations
Figure 10-19

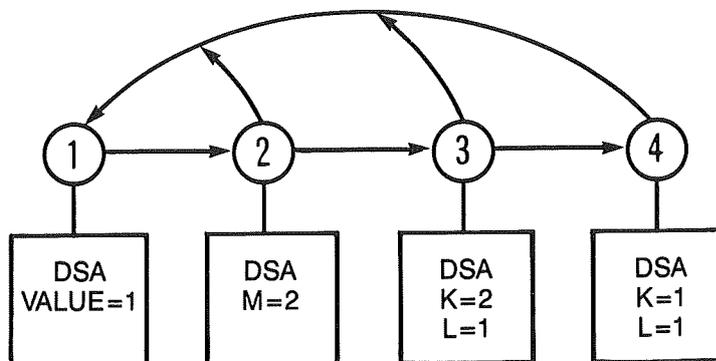
We use the convention that the circle representing a block invocation will point back to the circle representing its environmental block invocation. If we modify the diagrams in Figure 10-18 to use this convention, the result is as shown in Figure 10-19. Notice that, for example, in line 4 of this figure, there are three invocations of internal blocks, and for each of these, the environmental block invocation is block invocation number 1. For this reason, each of the circles representing block invocations 2, 3, and 4 points back to the circle representing block invocation number 1.

If a block invocation calls an external procedure, there is no environmental block invocation.

Representing the Dynamic Storage Area

One dynamic storage area (DSA) is allocated for each block invocation. The DSA contains the parameters and AUTOMATIC storage allocated for that block invocation.

In our dynamic diagram, we may indicate the DSA for each block invocation as a box attached to the circle that represents the block invocation. For example, consider line 4 of Figure 10-19. At the point where the RETURN statement in procedure C is executed, the diagram may be modified as shown in Figure 10-20. Notice that each of the block invocation circles has a DSA box attached to it, and the box shows the variables contained in the DSA. In the case where there are two or more invocations of the same recursive block, the DSAs for the different invocations will contain the same variables, possibly with different values. This is illustrated in Figure 10-20 in block invocation numbers 3 and 4.



Dynamic Storage Area (DSA) for Block Invocations
Figure 10-20

How LABEL Variables Are Implemented

A GO TO statement may terminate an active block. Such a termination is said to be an abnormal termination.

If the target of a GO TO statement is a LABEL constant, as is usually the case, then it is always clear to what block invocation the statement is transferring. If the LABEL constant is inside the block containing the GO TO statement, no block invocations are terminated. If the LABEL constant is outside the block that contains the GO TO statement, but is inside the immediately containing block, then the transfer is made to the environmental block invocation, and the block executing the GO TO statement is abnormally terminated. If the environmental block invocation is different from the invoking block invocation, all block invocations after the environmental block invocation are terminated.

In the case of a LABEL variable, the situation is a bit more complicated, since the value of the LABEL variable must specify a block invocation number. Consider the program illustrated in Figure 10-21.

```

S:  PROC OPTIONS(MAIN);
    DECLARE L(4) LABEL;
    K = 0;
    CALL R;
R:  PROC RECURSIVE;
    DECLARE V;
    K, V = K + 1;
    L(V) = LB;
    IF K < 4 THEN CALL R;
      ELSE CALL T;
LB:  PUT LIST(V);
    END R;
T:  PROC;
    DECLARE M;
    GET LIST(M);
    GO TO L(M);
    END T;
    END S;

```

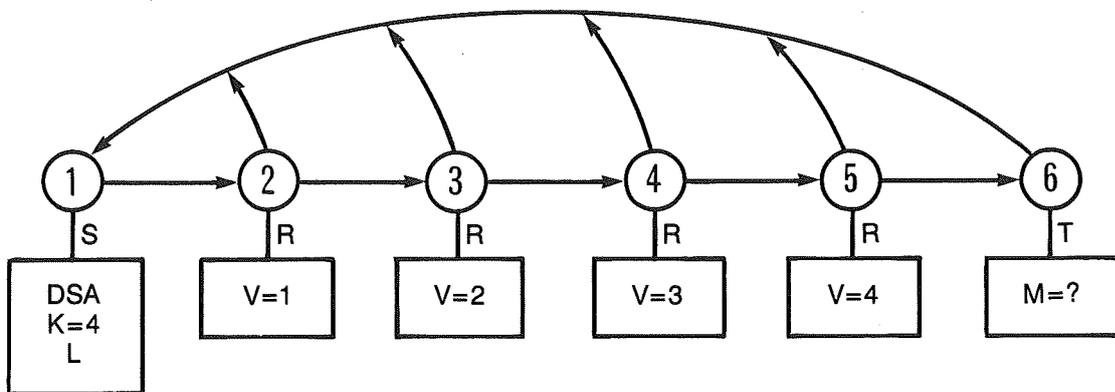
Label Variables for Blocks
Figure 10-21

If you trace through execution of this program, you see that it does the following:

1. The main program, block S, invokes block R.

2. Block R sets L(1) to the LABEL value LB. R then invokes itself recursively.
3. R sets L(2) to the LABEL value LB, and invokes itself recursively.
4. R sets L(3) to the LABEL value LB, and invokes itself recursively.
5. R sets L(4) to the LABEL value LB, and invokes block T.
6. Block T inputs a value for M and executes a GO TO statement, transferring control to the location specified by the LABEL variable L(M).

At the point just before the GO TO statement is executed, the dynamic block invocation structure is indicated by Figure 10-22. As this diagram shows, there are four simultaneous invocations of the recursive block R at this point.



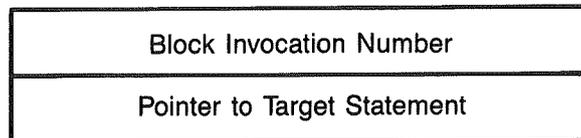
Invocations of a Recursive Procedure With LABEL Variables
Figure 10-22

All four elements of the LABEL variable array L point to the same statement, the one with LABEL LB. Therefore, the statement

GO TO L(M);

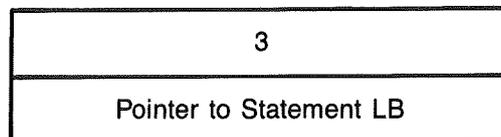
transfers to the statement with LABEL LB. The question is: to which invocation of block R does the GO TO statement transfer?

A LABEL variable value contains two pieces of information, as illustrated in Figure 10-23. It contains a block invocation number and a pointer to the statement to which the transfer is to be made. Since the assignments to the four different elements of the array L were made within four different invocations of block R, the four elements of array L will contain four different block invocation numbers. The GO TO statement returns to the invocation whose block number is specified in the value of LABEL variable L(M).

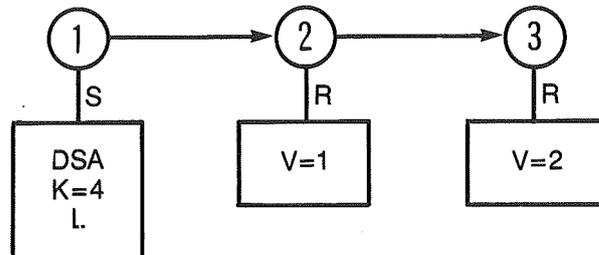


Information in a LABEL Variable
Figure 10-23

Suppose, for example, that the input value of M is 2. The value of L(2) is shown in Figure 10-24. Therefore, the GO TO statement returns to block invocation number 3. The dynamic block invocation chain diagram becomes the diagram shown in Figure 10-25. That is, block invocation numbers 4, 5 and 6 are all abnormally terminated. The PUT LIST statement prints the value 7.



Return to Block Invocation 3 (LABEL Variable = 2)
Figure 10-24



Abnormal Termination of Blocks 4, 5, 6
(Result of Preceding Figure)

Figure 10-25

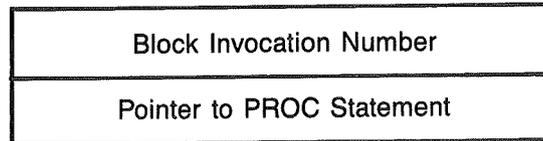
A final note: although our example program does not illustrate it, the statement

```
GO TO L(4);
```

would now be illegal. That is, it would be legal to transfer to the statement with LABEL LB, but not within the block invocation specified by the number in the value of L(4), since that block invocation has been terminated.

How ENTRY Variables Are Implemented

Like LABEL variables, ENTRY variables contain two pieces of information, as illustrated in Figure 10-26. In this case, however, the block invocation number is the number of the environmental block invocation to be used when the procedure is invoked.



Information in an ENTRY Variable
Figure 10-26

Consider the program illustrated in Figure 10-27.

```

M: PROCEDURE OPTIONS(MAIN);
  DECLARE E ENTRY VARIABLE;
  I = 0;
  CALL R;
R:   PROC RECURSIVE;
  DECLARE K;
  K, I, = I + 1;
  IF K = 2 THEN E = S;
  IF K < 4 THEN CALL R;
  ELSE CALL T;
S:   PROC;
  PUT LIST(K);
  END S;
  END R;
T:   PROC;
  CALL E;
  END T;
  END M;

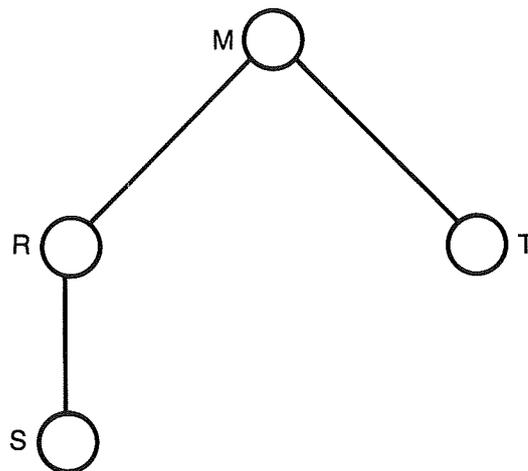
```

A Program With ENTRY Variables
Figure 10-27

The static block structure tree for this program is shown in Figure 10-28. This program executes as follows:

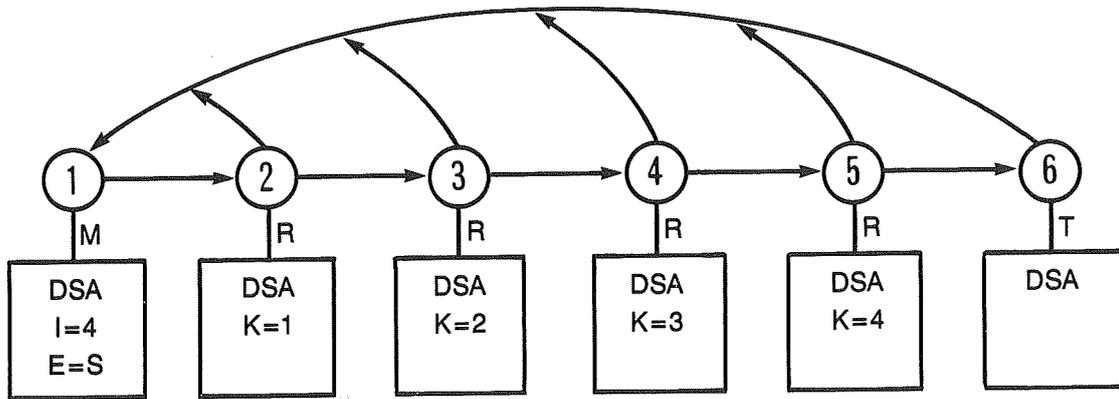
1. The main program block, block M, invokes block R.
2. R invokes itself recursively.
3. R sets the ENTRY variable to S, and then invokes itself recursively.

4. R invokes itself recursively.
5. R invokes block T.
6. T executes a CALL statement specifying the ENTRY variable E.



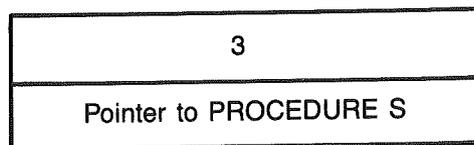
Static Structure Diagram for ENTRY Variables
Figure 10-28

At the point where block T executes the CALL statement, the dynamic block structure is as shown in Figure 10-29. There are four invocations of the recursive block R, as shown in that diagram. Since E was set equal to the entry point S, the statement CALL E invokes the procedure S. The question is: what is the environmental block invocation number?



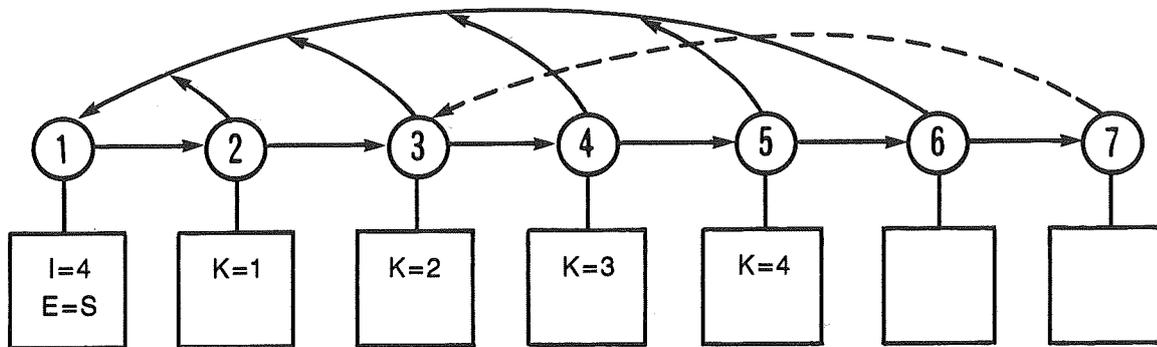
Dynamic Structure for ENTRY Variables
Figure 10-29

Notice that the value of E was set only once, inside block invocation number 3, where the value of K equals 2. That is, the value of the ENTRY variable E is as shown in Figure 10-30. Therefore, the statement CALL E invokes block S, and the environmental block invocation number is 3.



Result of CALL E in Figure 10-27
Figure 10-30

Therefore, when S is invoked, the dynamic block structure becomes what is shown in Figure 10-31. Notice that circle number 7 points back to circle number 3 since the environmental block invocation for invocation number 7 is number 3. Therefore, the PUT LIST statement in S prints the value K of 2 since the value of K is inherited from the environmental block invocation number 3.



Result of Invocation of Block S in Figure 10-27
Figure 10-31

Rules for Block Invocation and Termination

We now summarize some rules for block invocation and termination. Examples of these rules have already been given in the preceding sections.

1. If a block invocation terminates normally, control returns to the invoking block invocation.
2. If a block invocation terminates abnormally, any number of additional block invocations may simultaneously terminate abnormally. By appropriate use of LABEL variables, control may be returned to any active block invocation, and all subsequent block invocations terminate abnormally at the same time.
3. By appropriate use of ENTRY variables, any PROCEDURE block contained immediately in any active block may be invoked. If the PROCEDURE block is contained immediately in a recursively active block, the procedure may be invoked in such a way that any of the recursive invocations of the containing block may become the environmental block invocation.

There is a special case of the third rule: any external procedure may be invoked from any other block, subject to the restriction that a nonrecursive block may not be recursively invoked.

Implementation of On-units

The way PL/I handles on-units and on-unit invocation is particularly tricky when recursive procedures are involved. On-units are introduced

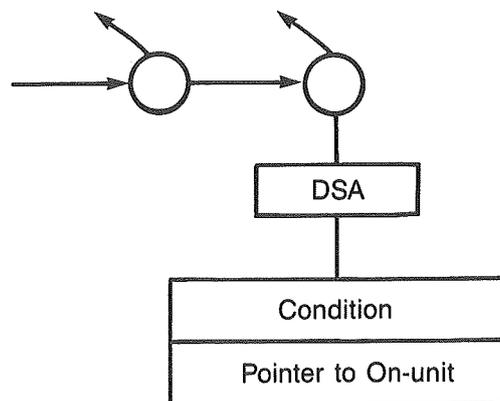
at the end of Chapter 4. Before reading the present section, however, be familiar with on-units as presented in Chapter 13.

This section outlines the actions that PL/I takes for the following statements and operations:

- ON condition on-unit;
- ON condition SYSTEM;
- REVERT condition;
- raising a condition.

We explain these actions by showing how each of these actions affects the diagram of the dynamic block invocation chain.

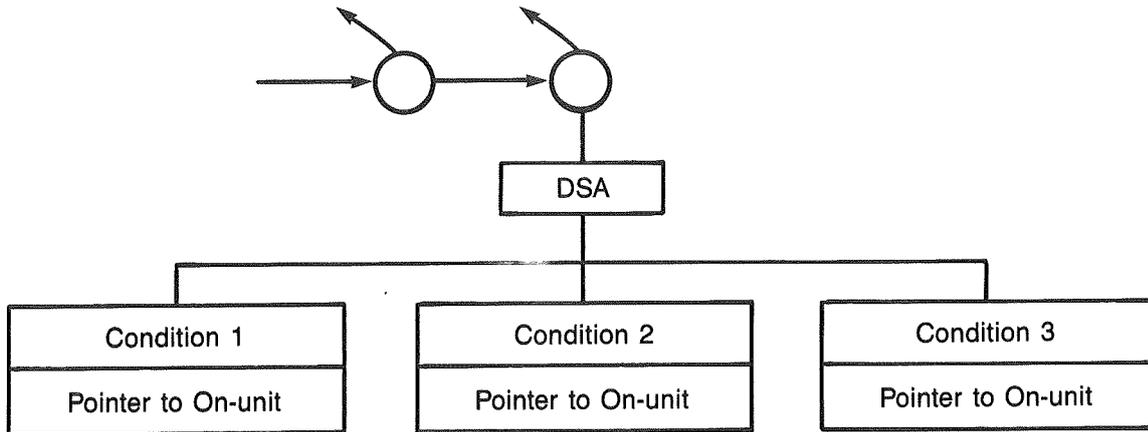
Let's begin with the execution of an ON statement where an on-unit is established. Figure 10-32 illustrates how an established on-unit is represented in the dynamic diagram. In the block invocation that executes the ON statement that establishes the on-unit, the diagram is modified so that there is an on-condition box attached to the DSA. This on-condition box contains the name of the condition and a pointer to the established on-unit. This on-condition box is added only when the ON statement is actually executed.



Dynamic Representation of an On-unit
Figure 10-32

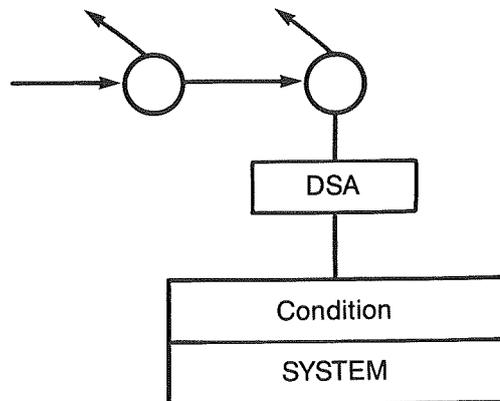
There may be several of these on-condition boxes attached to a single DSA for a single block invocation. This would happen, for example, if several ON statements for different conditions are executed during the same block invocation. In this case, the result might be as illustrated in Figure 10-33, which shows three on-condition boxes

attached to the same DSA. They represent established on-units for three different conditions, as the result of the execution of three ON statements during the same block invocation.



Multiple On-units for One Block Invocation
Figure 10-33

If a block invocation executes an ON statement specifying the SYSTEM option, an on-condition box is still attached to the DSA, but the box does not contain a pointer to the established on-unit; instead, it contains the word SYSTEM to indicate the SYSTEM option. This is illustrated in Figure 10-34.



An On-unit for the SYSTEM Option
Figure 10-34

Now suppose a block invocation executes two ON statements specifying the same condition. This is reflected in the dynamic diagram as follows: when the second ON statement executes, the new on-condition box replaces the one created when the first ON statement was executed. Therefore, there can never be more than one on-condition box for the same condition attached to the same DSA.

When a block invocation executes a REVERT statement for a condition, the diagram is modified by entirely removing any on-condition box for that condition from the DSA for the current block invocation.

Now let us suppose that a program error occurs and it is necessary to raise a condition and either take the standard system action for the condition or invoke an established on-unit. In terms of the dynamic block invocation chain diagram, PL/I proceeds as follows:

1. Starting from the current block invocation, and continuing back through the invoking block invocations, it finds the most recently created on-condition box for the desired condition.
2. If no on-condition box is found for the desired condition, or if the most recently created box specifies the SYSTEM option, then it takes the standard system action for the condition; otherwise, it proceeds with the next step.
3. It invokes the established on-unit specified by the on-condition box. Note that this is a new block invocation. The environmental block invocation is the new block invocation for the invocation that contained the on-condition box.

The following example illustrates how these rules work.

```

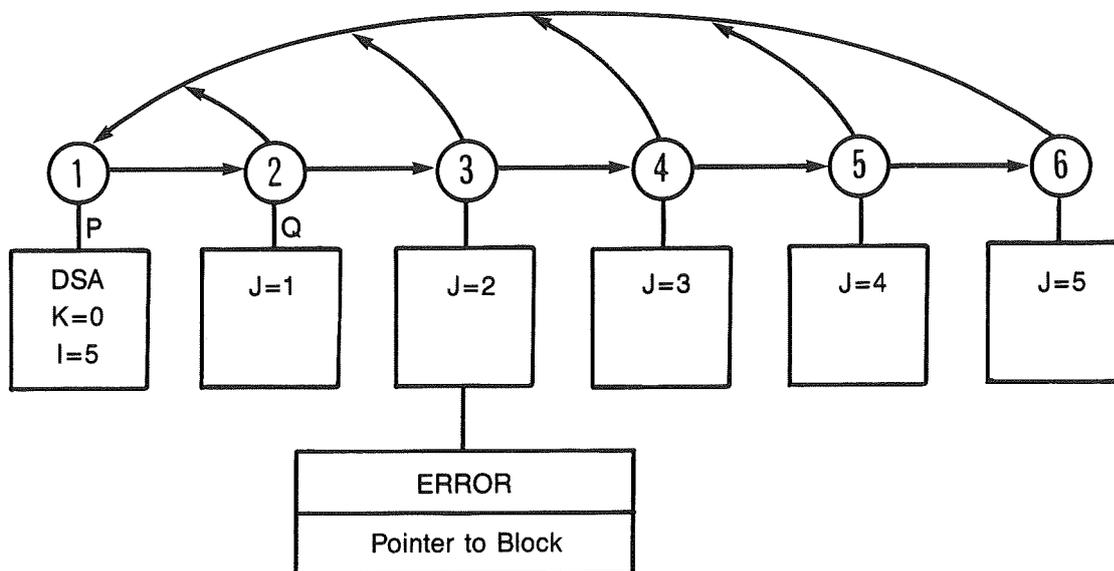
P:    PROC OPTIONS(MAIN);
      DECLARE (I, K) INITIAL(0);
      CALL Q;
Q:    PROC RECURSIVE;
      DECLARE J;
      J, I = I + 1;
      IF J = 2
        THEN ON ERROR
X:          BEGIN;
            K = J;
            GO TO EXIT;
            END;
      IF J < 5 THEN CALL Q;
      PUT LIST(LOG(-1))/*RAISE ERROR*/;
      RETURN;
      END Q;
EXIT:  PUT LIST(K);
      END P;

```

Execution of this program proceeds as follows:

1. Procedure P invokes procedure Q.
2. Procedure Q sets J equal to 1 and invokes Q recursively.
3. Q sets J equal to 2, establishes an ERROR on-unit, and invokes Q recursively.
4. Q sets J equal to 3, and invokes Q recursively.
5. Q sets J equal to 4, and invokes Q recursively.
6. Q sets J equal to 5 and then executes a statement that raises the ERROR condition. Figure 10-35 shows the dynamic diagram at the point just before the ERROR condition is raised. Notice that block invocation number 3 contains an on-condition box for the ERROR condition, since that is the only invocation that executed an ON statement.

As a result, block X, the on-unit established for the ERROR condition, is invoked with environmental block invocation number 3.

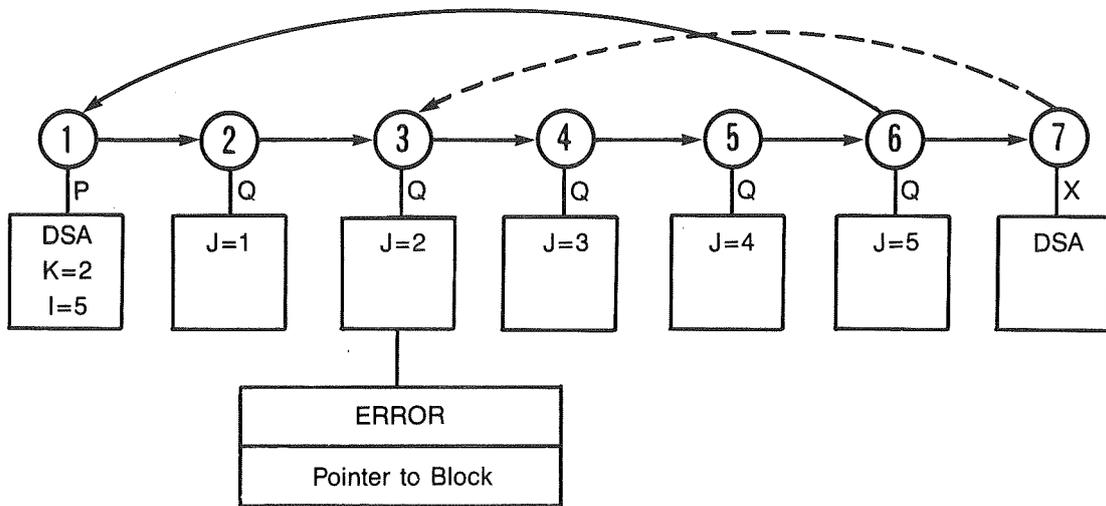


The Dynamic Block Chain before an Error Condition Occurs
Figure 10-35

7. Block X sets K equal to J. Since the environmental block invocation is invocation number 3, the value of J in that block invocation is used.

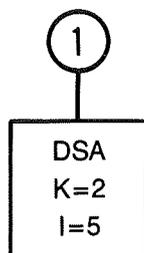
As a result, K is set equal to the value 2. At that point, the dynamic diagram is as shown in Figure 10-36. Notice that, in the DSA for block invocation number 1, the value of 2 for K is indicated.

Block X then executes a GO TO statement to the label EXIT.



The Dynamic Block Chain With an Error Condition
Figure 10-36

This last action abnormally terminates block invocations 2 through 7, so the dynamic diagram becomes that shown in Figure 10-37, and the value 2 is printed.



Abnormal Termination After an Error Condition
Figure 10-37

COMPILER-DIRECTING STATEMENTS

Five statements, %INCLUDE, %REPLACE, %PAGE, %LIST, and %NOLIST, direct the compiler to alter the text of the source file or to change the listing format.

During the compilation of a program module, the compiler recognizes and evaluates two statements that alter the program text. These statements, %INCLUDE and %REPLACE, simplify the job of writing large programs, but are also useful in small programs.

The general form of %INCLUDE is

```
%INCLUDE 'filename';
```

where filename is the name of a text file that is to be inserted into the program text in place of the %INCLUDE statement. The filename is a PRIMOS filename.

%INCLUDE can appear in place of a name, constant, or punctuation symbol. The included text may contain additional %INCLUDE statements, but normally contains declarations that are common to more than one program module.

If the file named in a %INCLUDE statement also contains a %INCLUDE statement, the second %INCLUDE statement is said to be nested. The maximum number of nestings of %INCLUDE statements is 32.

The general form of %REPLACE is

```
%REPLACE name BY [-]constant [, name BY [-] constant] ...;
```

Each occurrence of name that follows the %REPLACE is replaced by the constant or signed constant. %REPLACE normally is used to supply the sizes of tables or to give names to special constants whose meaning would not otherwise be obvious, as in this example:

```
%REPLACE TRUE BY '1'B;
%REPLACE TABLE_SIZE BY 100;
%REPLACE MOTOR_POOL BY 5;

. . .
DECLARE X(TABLE_SIZE) FIXED STATIC;

. . .
DO K = 1 TO TABLE_SIZE;

. . .
IF DEPARTMENT_NUMBER = MOTOR_POOL
THEN DO;

. . .
```

Both %REPLACE and %INCLUDE operate on the program text without regard to the meaning of the text. Thus the replaced name can accidentally be a keyword such as STOP or READ. In this case, an "unrecognizable statement" error message is issued by the compiler when it reads a subsequent STOP or READ statement. %REPLACE replaces all subsequent occurrences of name without regard to the block structure of the module.

A third compiler-directing statement, %PAGE, causes the compiler listing, if any, to skip to a new page.

The last two compiler-directing statements, %NOLIST and %LIST, suppress and restart the printing of the source listing. Program statements between a %NOLIST statement and a %LIST statement do not appear in the source listing file.

Note

%INCLUDE, %REPLACE, %PAGE, %LIST, and %NOLIST are Prime extensions to standard PL/I and may not be available in other implementations of PL/I.

%INCLUDE Files and the Search Rules Facility

As of Rev. 21.0, the PRIMOS search rules facility enables you to establish an INCLUDE\$ search list. An INCLUDE\$ search list is a list of directories that are to be searched for a %INCLUDE file whenever a %INCLUDE statement is processed. (Although there are several kinds of search lists, this section explains only the INCLUDE\$ search list. For complete information about the PRIMOS search rules facility, see the Advanced Programmer's Guide, Volume II.)

When you specify a file in a %INCLUDE statement, you must ordinarily

give as much of the file pathname as PRIMOS needs to locate the file. If you use %INCLUDE files often, and if the files are kept in a number of different directories, keeping track of the file pathnames can be difficult. Now, however, you can locate %INCLUDE files by supplying only a filename and using the search rules facility to provide the full pathname.

Establishing Search Rules: To establish search rules for %INCLUDE files, perform the following steps:

1. Create a template file called

[yourchoice.]INCLUDE\$.SR

This file should contain a list of the pathnames of the directories that contain your %INCLUDE files. For example, you might create a file called MY.INCLUDE\$.SR that contains the following directory names:

```
<SYS1>MASTER_DIR>INSERT_FILES
<SYS2>ME
```

2. Activate the template file by using the SET_SEARCH_RULES (SSR) command. For example, if your template file is named MY.INCLUDE\$.SR, type

OK, SSR MY.INCLUDE\$

This command sets the INCLUDE\$ search list for your process. This search list may contain system search rules and administrator search rules in addition to the rules you specified in MY.INCLUDE\$.SR. If you will use a search list often, you should include the command to activate the list in your LOGIN.CPL file.

When you give the SSR command shown in step 2, PRIMOS copies the contents of MY.INCLUDE\$.SR into your INCLUDE\$ search list. If you have no special system or administrator search rules, your INCLUDE\$ search list appears as follows when you give the LIST_SEARCH_RULES (LSR) command:

```
List: INCLUDE$
Pathname of template: <MYSYS>ME>PL1>MY.INCLUDE$.SR
```

```
[home_dir]
<SYS1>MASTER_DIR>INSERT_FILES
<SYS2>ME
```

[home_dir], your current attach point, is the system default. It is always the first directory searched, unless you remove it from the list or change the order of evaluation by using the `-NO_SYSTEM` option of the `SSR` command. Additional search rules, established as system-wide defaults by your system administrator, may also appear at the beginning of your `INCLUDE$` search list.

The `SET_SEARCH_RULES` and `LIST_SEARCH_RULES` commands are described in the PRIMOS Commands Reference Guide. For more information about establishing search rules, see the Advanced Programmer's Guide, Volume II.

Using Search Rules: Once you have set the search list, any `%INCLUDE` statement in a program can give just the filename rather than the full pathname of the file. PRIMOS then searches the contents of the directories in the `INCLUDE$` search list for the filename specified in the `%INCLUDE` statement. If PRIMOS finds the file, it stops searching and returns the full pathname of the file to the compiler. The compiler then uses this pathname to locate the file and inserts its contents into the source program.

Using [referencing_dir]: The Advanced Programmer's Guide describes several expressions that you can use in your list of search rules. One of these, [referencing_dir], has a special meaning for `INCLUDE$` search lists. Like [home_dir], [referencing_dir] is a variable that PRIMOS replaces with a directory pathname. [referencing_dir] always evaluates to the pathname of the directory from which the request for a `%INCLUDE` file is made. Thus, if a `%INCLUDE` statement is located in a program, [referencing_dir] evaluates to the pathname of the directory that contains the program.

[referencing_dir] is useful if the following three situations exist.

- You are compiling a program that is not in your current directory.
- The directory containing the program is not in your search rules list.
- The program contains a `%INCLUDE` statement.

Under the above circumstances, the search for the `%INCLUDE` file succeeds only if [referencing_dir] is in your list of search rules.

You can also use [referencing_dir] for programs that contain nested `%INCLUDE` statements. `%INCLUDE` statements are nested if the file requested by one `%INCLUDE` statement also contains one or more `%INCLUDE` statements. If nested `%INCLUDE` statements request files located in the same directory as the `%INCLUDE` file in which they are nested, putting [referencing_dir] at the top of your search rules list could speed up the search somewhat.

11

STREAM

Input/Output

The PL/I language supports two methods for doing input/output: `STREAM I/O` and `RECORD I/O`. This chapter discusses `STREAM I/O`. `STREAM I/O` uses `GET` and `PUT` statements, which treat a file as a stream of characters. This method preserves machine independence. `STREAM I/O` is also used for terminal input/output. More explanation of the differences between `STREAM` and `RECORD I/O` is provided in the section on `STREAM INPUT/OUTPUT SPECIFICATIONS` later in this chapter.

The statements discussed are

- The `PUT` statement, for general output.
- The `GET` statement, for general input.
- The `WRITE` statement, for variable-length uninterpreted output (on Prime systems only).
- The `READ` statement, for variable-length uninterpreted input (on Prime systems only).
- The `FORMAT` statement, which allows you to specify a common format list for several `PUT EDIT` or `GET EDIT` statements.
- The `DECLARE` statement, which allows you to specify that a certain identifier is to have the `FILE` attribute. You need this and the `OPEN` statement in order to do input from or output to all files and devices.

- The OPEN statement, which allows you to connect your program's FILE identifier to an external file or device.
- ON ENDFILE and ON ENDPAGE.

The DECLARE and OPEN statements for files are discussed in greater detail in Chapter 12, RECORD INPUT/OUTPUT.

INTRODUCTION TO THE PUT STATEMENT

Previous chapters contain a number of examples of the PUT statement. This section summarizes the most commonly used PUT statement options. Detailed specifications for the PUT statement can be found later in this chapter.

The syntax for the PUT statement is

PUT option-list;

The option-list is a list of options separated by blanks. The most commonly-used options are

- FILE(reference): See the section PUT AND GET TO FILES AND DEVICES.
- STRING(reference): See The PUT STRING Statement.
- SKIP or SKIP(1): PL/I skips to the next output line (or output record) before doing any further output.
- SKIP(n), where $n > 1$: PL/I skips to the next output line (or record), then skips $(n - 1)$ additional blank lines (or records).
- PAGE: PL/I skips to a new output page.
- LINE(n): PL/I moves to output line n on the printer page, skipping to a new page if necessary.
- LIST, DATA, EDIT: These are described in this section.

If you specify neither the FILE nor the STRING option in your PUT statement, PL/I assumes FILE(SYSPRINT), output to the terminal. You must specify at least one of the options SKIP, PAGE, LINE, LIST, DATA, and EDIT. You may specify at most one of the options SKIP, LINE, and PAGE, and at most one of the options LIST, EDIT, and DATA. You may specify the options in any order.

Here are some examples:

```
PUT PAGE;  
  
PUT LIST(X, Y + 3);  
  
PUT FILE(TAPEOUT) SKIP EDIT(A + B) (F(5));  
  
PUT SKIP LIST(X);  
  
PUT LIST(X) SKIP;
```

The first of these statements causes PL/I to skip to the top of a new printer page. The second prints the values of X and Y + 3. The third specifies a FILE identifier called TAPEOUT and specifies that PL/I is to skip to a new output record and then print the value of A + B in the format F(5). The last two statements have exactly the same effect and are included here to illustrate the fact that the options may go in any order. Many beginning programmers believe that the last statement skips to a new output record after the value of X is printed, while, in fact, the effect of the SKIP option always takes place before any further output occurs.

Note that unlike other languages, PL/I does not automatically execute a SKIP option at the end of output from a PUT statement. Therefore, the two statements

```
PUT LIST(A);  
  
PUT LIST(B);
```

taken together, have exactly the same effect as the single statement

```
PUT LIST(A, B);
```

In both cases, output appears on the same line. To avoid this, use the two statements

```
PUT LIST(A);  
  
PUT SKIP LIST(B);
```

The SKIP option in the second of these statements causes the value of B to be printed on a new line.

The PUT LIST Statement

Most of the examples of the PUT statements that appear in other chapters of this manual are for PUT LIST. You may use the LIST option to print the values of one or more expressions. For example, the statement

```
PUT LIST(A, X + Y, U*V);
```

prints the values of the three expressions within the set of parentheses following the word LIST.

An expression may be an aggregate expression. For example,

```
DECLARE AR(100) FIXED;  
...  
PUT LIST(AR + 5);
```

prints 100 values. Each of these 100 values is obtained by adding 5 to the corresponding element of the array AR. The complete set of rules for evaluating aggregate expressions is discussed in Chapter 6, EVALUATING EXPRESSIONS.

For each of the individual scalar data values appearing with the LIST option, PL/I does the following:

1. Prints sufficient blank characters to advance the output line to a tab stop. The tab stop positions are preset by the system, but they may be changed by the TABS option of the OPEN statement, as described later in this chapter.
2. Converts the scalar data items to the CHARACTER data type as described in Chapter 6.
3. Prints out the value of that CHARACTER string value.

The above rules apply to files intended to be printed on the terminal or line printer. Output files that are to be stored on disk have slightly different rules from those above.

The PUT DATA Statement

When you use DATA directed output, PL/I prints the variable name along with the value of the variable, in the format of an assignment statement. For example, the statement

```
PUT DATA(M, N);
```

would cause PL/I to print something like the following:

```
M =      5           N =      20;
```

If the variable being printed is an aggregate, PL/I prints all the individual scalar elements of the aggregate. For example, consider the following statements:

```
DECLARE A(5) FLOAT DECIMAL(5);
...
PUT SKIP DATA(A);
```

These statement would print something like the following:

```
A(1)= 2.8734E+02      A(2)= -7.4628E-05      A(3)= 1.0000E+00
A(4)= 8.6500E+02      A(5)= 0.0000E+00;
```

Notice that each scalar value in the aggregate is printed, as in the case of PUT LIST, but PUT DATA prints additional information, the name of the variable or aggregate element. After the last variable/value pair is printed, the output ends with a semicolon.

The use of PUT DATA has advantages and disadvantages. The major advantages are that you clearly see which variable is associated with each value being printed; this can be very useful during the debugging stages of your program, or for printing dumps of program variables after unexpected errors. The main disadvantage of PUT DATA is the cluttered appearance of the output, which makes it unpleasant to read.

One form of the PUT DATA statement is particularly useful in certain debugging situations. The statement

```
PUT DATA;
```

where you supply no list of variables with the DATA option, causes PL/I to print the name and value of all variables in your program accessible from the PUT statement. The result is usually several pages of very cluttered output, but this can be useful in certain debugging situations.

Unlike PUT LIST, PUT DATA does not allow arbitrary expressions in the list following the DATA keyword. For example, the statement

```
PUT DATA(X + Y);
```

is illegal.

The PUT EDIT Statement

Let us now turn our attention to the formats of the output that PL/I prints. In the case of PUT LIST and PUT DATA, PL/I prints the data values in an output format that is determined by the data type of the values. For example, if the value is FIXED, it is printed as a decimal number, possibly with a decimal point. If the data value is FLOAT, it is printed in scientific notation.

PUT EDIT permits you to specify not only the value to be printed, but also the format in which the value is to be printed. PUT EDIT also gives you complete control over both horizontal and vertical spacing of your printed data, so that you can get columns, tables, graphs, and other printouts that are pleasing to the eye.

Consider the following example:

```
DECLARE (X, Y) FLOAT;  
X = 8.3;  
Y = -65.32;  
PUT EDIT(X, Y) (F(4), F(9,3));
```

Look at the PUT EDIT statement, the last statement in the above example. It contains a list of data values (the variables X and Y whose values are to be printed) but the statement also contains a format list, containing the two format items, F(4) and F(9,3). The result is the following:

1. PL/I prints the value of X in the format F(4). This format specifies that the output value, no matter what its data type, is to be printed as an integer in a four-character field. Therefore, since the value of X is 8.3, it is printed as bbb8, where b is a blank character.
2. The value of Y is -64.32, and PL/I prints this value in the format F(9,3), a format item that specifies a nine-character output field, with three digits after the decimal point. The result is that the value of Y is printed as bb-64.320. Notice that there are two leading blanks among the characters printed, in order to fill out the results to nine characters.

PUT EDIT differs from PUT LIST in that you are completely in control of the spacing between output data items. For PUT LIST, PL/I automatically inserts blank characters between output data items; but for PUT EDIT, PL/I only prints those blank characters called for by your format list. Therefore, in the above example of a PUT EDIT statement, the full output is

bbb8bb-64.320

There are exactly five blank characters in this output, as called for by the format items.

Many beginning programmers confuse the format item F(8,3) with the numeric data type FIXED DECIMAL(8,3). Although there are some similarities, there are also very important differences. FIXED DECIMAL(8,3) is a data type for a variable that can have a sign (plus or minus) and eight digits, three of which follow the decimal point. F(8,3) is a format item specifying output of eight characters, including three digits after the decimal point. These eight characters include the printed sign, if any, and the decimal point. The largest number that can be printed in the format F(8,3) is

9999.999

while the smallest negative number is

-999.999

This contrasts to the fact that a variable with the data type FIXED DECIMAL(8,3) can have a value as large as

+99999.999

and a negative value as small as

-99999.999

Thus, the width in the format items specifies the total number of characters, including the sign, if any, and the decimal points, if any. The precision in the data type specifies only the number of digits, and does not include the sign or the decimal point. Therefore, the F(8,3) format item is not always sufficiently wide to accommodate the value of a FIXED DECIMAL(8,3) variable.

Common Data Formats: Later in this chapter, there is a section called DETAILED SPECIFICATIONS FOR THE PUT STATEMENT. That section describes all the PUT EDIT format items and defines them precisely. The following lines list the most commonly used PUT EDIT format items for data values. Table 11-1 illustrates each of these format items.

- F(w): PL/I prints the data value as a decimal integer, in a field of w characters.
- F(w,d): PL/I prints the data value as a decimal number with a decimal point, in a field of w characters, with d digits following the decimal point. If d = 0, no decimal point is printed.
- E(w,d): PL/I prints the data value in scientific notation, with the mantissa and characteristic fields separated by the letter E, with d digits following the decimal point and one digit preceding the decimal point in the mantissa. The total field size, including both mantissa and characteristic, is w characters.
- 'picture': PL/I prints the data value in the format determined by the CHARACTER string value of the picture specification. See Chapter 5 for details on picture specifications.
- A: PL/I prints the data value as a CHARACTER value. All characters in the data value are printed.
- A(w): PL/I prints the data value as a CHARACTER value in a field of w characters. If the data value contains fewer than w characters, PL/I prints out additional blank characters on the right, to pad out the data value. If the data value contains more than w characters, PL/I truncates it and prints only the first w characters.
- C(format, format): PL/I prints the data value as a COMPLEX value. PL/I prints the real part of the COMPLEX value using the first format, and prints the imaginary part of the data value using the second format.

Each of these format items is illustrated in Table 11-1. Additional illustrations of each format item are given in the section on DETAILED SPECIFICATIONS FOR THE PUT STATEMENT later in this chapter.

Table 11-1
Common PUT EDIT Format Items

Format	Data Value	Characters Printed
F(4)	23	bb23
F(5)	23	bbb23
F(5)	-23	bb-23
F(5,1)	23	b23.0
F(8,2)	-7.924	bbb-7.92
E(10,4)	92.4	9.2400E+01
P'9999'	23	0023
A	'ABCD'	ABCD
A(5)	'ABCD'	ABCDb
A(3)	'ABCD'	ABC
C(F(4),F(7,1))	45+23I	bb45bbb23.0

Control Formats: So far in this chapter we have discussed some of the data format items. These are format items that specify the format in which a particular data value is to be printed. PL/I provides another set of format items called the control format items. Use these format items to specify the position of the output; that is, specify where in the output line or output record the data is to appear, or where on the printer page it is to appear. For example, if you wish your data items to appear in columns, with headings at the top of the printer page, use control format items to control the spacing between the data items printed.

Here is an example:

```
X = 3;
PUT EDIT(X, 2 * X + 1) (F(4),X(3),F(3));
```

In this PUT EDIT statement, there are two data values to be printed, and the format list contains three format items. The first and last of these are data format items, in that they specify the format in which each of the two data values is to be printed. The second format item, X(3), is a control format item. It does not specify how any data value is to be printed, but it does specify that three additional blank characters are to be inserted into the output. The result is that the output from the PUT statement would be

```

bbb3 bbb bb7
  F(4)  X(3)  F(3)

```

The F(4) and F(3) format items specify the printing of four and three characters, respectively. Because of the X(3) format item, three additional blank characters are printed.

The following is a list of the most commonly used control format items:

- X(w): PL/I prints w blank characters.
- COLUMN(n): PL/I moves to column n on the output line or output record. This means that PL/I prints as many blank characters as are necessary so that the next output value appears starting in column n. If output is already past column n on the current output record or output line, PL/I skips to a new line and prints (n - 1) blank characters.
- SKIP or SKIP(1): PL/I skips to the next output line or output record.
- SKIP(n), where n > 1: PL/I skips to the next output line, and then skips (n - 1) blank lines or records.
- PAGE: PL/I skips to the top of a new page in the output file.
- LINE(n): PL/I moves to output line n on the printer page, skipping to a new page if necessary.

Notice that the SKIP, PAGE, and LINE format items are handled exactly the same as for the corresponding options of the PUT statement.

The following example uses the X and SKIP format items:

```
A = 864;
B = 92;
PUT EDIT(A, B) (SKIP, F(5), SKIP, X(1), F(2));
```

The above PUT EDIT statement specifies two data values, A and B, to be printed, and the format list contains two data format items as well as three control format items. The output from this PUT statement is

```

      F(5)
  _____
b  b  8  6  4

  b  9  2
  _____
X(1) F(2)
```

Notice that the output is printed on two separate lines because of the SKIP format item, and that the blank character on the second line of output comes from the X(1) format item. The digits 92 on the second line of output come from the F(2) data format item.

If, in executing a PUT EDIT statement, PL/I finds that there are not enough format items in the format list to print all the data values in the data value list, PL/I restarts the format list from the beginning. For example, in the program segment:

```
A = 25;
B = 38.3;
PUT EDIT(A, B, A + B) (SKIP, F(5,2));
```

the PUT statement above contains three data values in the data value list, but only one data format item in the format list. As a result, the format list is restarted for each data value, to get the following results:

```
25.00
38.30
63.30
```

The control format item SKIP and the data format item F(5,2) are repeated for each of the three data values.

This feature of restarting the format list is particularly useful for printing an array. For example,

```
DECLARE A(200,3);
...
PUT EDIT(A) (SKIP, F(4), F(10,2), F(10));
```

In this example, the data list specifies the variable A, and so all 600 scalar elements of the array A are to be printed. The format list contains only three data format items, F(4), F(10,2), and F(10). The result is that the format list must be started from the beginning 200 times in order to print all 600 data values. Since the SKIP control format item is executed each time the format list is restarted, PL/I prints the array A in three columns, in a table containing 200 lines. The first column is printed in the format F(4), the second column in the format F(10,2), and the third column in the format F(10).

If you wish your format list to repeat a group of format items several times, you may use a repetition factor in your format list. The syntax is

```
n(format list)
```

Here you are specifying that the format list is to be repeated n times. Consider, for example, the following:

```
DECLARE B(10);  
...  
PUT EDIT(B) (3(F(5)),7(SKIP, F(10)));
```

This PUT statement prints the ten scalar elements of the array B. The first three elements of B are printed using the format F(5), which is to be used three times as specified by the repetition factor, and the last seven elements of B are to be printed using the format list SKIP, F(10).

The PUT STRING Statement

The PUT statement prints the values of variables on the terminal, on a printer, or stores them in a file. Part of PL/I's job in executing a PUT statement is to convert the value of a variable from internal encoded machine format into a string of characters that can include digits, decimal points, and signs.

Sometimes you wish to take advantage of this conversion process without actually doing any output to a device external to your program. In PL/I terminology, this is the wish to convert a numeric value into a CHARACTER format. One way to do this is to do an implicit numeric to string conversion in an assignment statement. For example, consider the following:

```
DECLARE X FLOAT DECIMAL(6);  
DECLARE C CHARACTER(100) VARYING;  
...  
X = 25.3;  
C = X;
```

The assignment statement at the end of this example causes PL/I to convert implicitly the numeric value of X to a CHARACTER string value and assign it to C with the result that C = 'b2.53000E+01'. However, this kind of implicit conversion gives you no control over the format in which the variable is to be stored in CHARACTER form.

By means of the `STRING` option, PL/I allows you to use the full power of the `PUT` statement in converting numeric data to `CHARACTER` data in whatever format you wish. For example, you can change the preceding example to the following:

```
DECLARE X FLOAT DECIMAL(6);
DECLARE C CHARACTER(100) VARYING;
...
X = 25.3;
PUT STRING(C) EDIT(X) (F(7,2));
```

This last statement is similar in effect to the final assignment statement of the preceding example in that in both cases PL/I converts the value of the numeric value `X` into character form. In the case of this `PUT` statement, since a `STRING` option is specified, PL/I does not do output to a device or file. Instead, PL/I stores the output characters in the variable `C`. The result is that `C` equals 'bb25.30', where the value of `X` was converted to `CHARACTER` format and stored into the variable `C`, using the format item `F(7,2)`.

The general syntax of this option is

```
PUT STRING(variable) other-options;
```

When you are using `PUT STRING`, usually you use the `EDIT` option, since you want to control the format explicitly. However, the `LIST` and `DATA` options are also legal.

The following options and format items are illegal with the `STRING` option: `FILE`, `SKIP`, `PAGE`, `LINE`, and `COLUMN`.

Notice that `PUT STRING` is no longer an output statement, since no actual data transmission to an external device is performed. Instead, `PUT STRING` is a purely computational statement.

INTRODUCTION TO THE GET STATEMENT

The syntax of the `GET` statement is

```
GET option-list;
```

This section introduces the most commonly used options of the `GET` statement. The section DETAILED SPECIFICATIONS FOR THE GET STATEMENT later in this chapter describes all the options in detail.

The most commonly used options are

- FILE(reference): See the section PUT AND GET TO FILES AND DEVICES.
- STRING(expression): See The GET STRING Statement.
- SKIP or SKIP(1): PL/I skips the remainder of the current input record and moves to the beginning of the next input record before doing any more input.
- SKIP(n), where $n > 1$: PL/I skips to the beginning of the next input record, and then skips $(n - 1)$ additional input records before doing any further input.
- COPY: PL/I copies all input characters to the standard output file (SYSPRINT).
- COPY(reference): PL/I copies all input characters to the output file specified by the reference, which must be to a FILE constant or expression.
- LIST, DATA, EDIT: These are described later in this section.

The GET statement takes input data values from the input stream. This means that, when it executes a GET statement, PL/I treats the input file or device as nothing more than a stream of characters, organized into records (or lines). Normally, record boundaries are ignored. The exceptions to this general principle are for those options and format items that specifically reference record boundaries. An example is the SKIP option, which causes PL/I to move to the beginning of the next input record.

In the GET statement, if you specify neither the FILE option nor the STRING option, PL/I assumes FILE(SYSIN), terminal input. You must specify at least one of the options SKIP, LIST, DATA, or EDIT. You may specify at most one of the options LIST, DATA, and EDIT. You may specify the options in any order.

Here are some examples:

```
GET LIST(X, Y);
```

```
GET FILE(TAPEIN) SKIP;
```

```
GET EDIT(A) (F(5));
```

For the first of these statements, PL/I takes two values from the input stream and assigns them to the variables X and Y. As we shall see, PL/I takes characters from the input stream until it finds an appropriate delimiter, either a blank or a comma. The second statement specifies that input is to be taken from a file called TAPEIN. This

GET statement specifies that you wish PL/I to skip the remainder of the current input record and to position further input at the beginning of the next record. The last statement specifies that five characters are to be taken from the input stream, and that these characters are to be converted to a numeric value and the result assigned to the variable A.

The GET LIST Statement

You may use the LIST option of the GET statement to input values for one or more variables. For example, the statement

```
GET LIST(A, X(K));
```

specifies that PL/I is to take values for the variable A and the array element X(K) from the input stream.

The input variable may be an aggregate. For example, consider these statements:

```
DECLARE AR(100) FIXED; DECLARE 1S, 2A, 2B;
...
GET LIST(AR);
GET LIST(S);
```

When executing the first GET statement, PL/I inputs 100 values from the input stream, converts them, and stores them in the array AR. The second GET statement inputs a value for each member of S in order, converts them, and stores them in the appropriate members.

The GET LIST statement takes characters from the input stream, ignoring boundaries between records. In the input stream, the individual data items should be separated by a comma, by one or more blank characters, or by both.

For instance, suppose that the input stream is coming from a disk file, and suppose that your program is executing the statement

```
GET LIST(A, B);
```

where A and B are numeric variables. Let us look at several examples of ways you might prepare an input stream file.

PL/I Reference Guide

You may type both values on one line, ending each one with a comma. For example, you might type

```
234,8764, CR
```

where CR stands for pressing the return key on your terminal. You may also end each data value with a blank, as in

```
234b8764b CR
```

You may also put the data onto separate input lines, which would be the case if you typed

```
234, CR  
8764, CR
```

Since PL/I simply ignores the return key in stream files, you may spread your input data over several lines, as would be the case if you typed the following:

```
2 CR  
34,8 CR  
764, CR
```

A common user error is to forget that a comma or blank must terminate each data value in the input stream. For example, if a file contained

```
234CR  
8764CR
```

PL/I would handle the above input as follows: PL/I would treat 2348764 as the first seven digits in the first input value, and then it would wait for more input, looking for a comma or a blank.

For input from the terminal, however, PL/I recognizes the return key as a terminator. The two lines just above are acceptable terminal input, while the three preceding them would be interpreted as four separate values. If you wish to disable recognition of the return key for terminal input as well as for stream files, use the `-ANSI` option. (See the section Interpretation of the TITLE Option Argument later in this chapter.)

Notice also that the above example could be changed by changing the GET statement into two statements,

```
GET LIST(A);  
GET LIST(B);
```

These two GET statements would have exactly the same effect in your PL/I program as the single GET statement illustrated above.

The data value taken from the input stream for a GET LIST statement may be any legitimate PL/I constant, always terminated by a blank or comma. If the input is a CHARACTER string value, it might look something like the following:

```
'STRING VALUE',
```

In this example, the input stream data value is a CHARACTER constant, followed here by a comma as terminator. Note that a CHARACTER constant containing a space or comma must be enclosed in single quotation marks in order to be interpreted as a single string. Examples of BIT and COMPLEX input values are

```
'1011101'B,  
23+4.5I,
```

Other legal constants of any computational data type are permitted. PL/I takes the constant and converts it to the data type of the variable in the GET LIST statement.

The longest acceptable terminal input stream value is 256 characters.

The GET DATA Statement

You may use GET DATA when you wish your input stream to contain the name of the variable as well as the value of the variable. To illustrate this, consider the statement

```
GET DATA(A, B, C, D, E);
```

PL/I Reference Guide

where A, B, C, D, and E are all scalar numeric variables. Let us take a look at some examples of what might be in the input stream where PL/I executes this statement. Depending upon what is in the input stream, PL/I may change or leave unchanged the values of any of the variables A, B, C, D, or E.

The input stream corresponding to a GET DATA statement contains what appears to be one or more consecutive assignments, only the last of which ends in a semicolon. The right-hand side of each assignment must be a constant following the same rules as for GET LIST.

For example, suppose that the input stream contains

```
A = 5, C = 23;
```

If PL/I finds these characters in the input stream when it executes the GET DATA statement shown above, PL/I changes the value of A to 5, changes the value of C to 23, and leaves the variables B, D, and E unchanged.

As another example, if the input stream corresponding to the GET DATA statement is

```
E = 43, B = 84, D = 92;
```

then PL/I sets the variables E, B, and D to the values shown above. This input stream example illustrates the fact that variables in the input stream need not appear in the same order as the variables in the GET DATA statement. You may separate the assignments in the input stream by a comma, or by one or more blanks, or both. Therefore, the input stream

```
E = 43      B = 84      ,      D = 92;
```

is handled by PL/I in exactly the same way as the input stream example above. PL/I keeps processing these assignments until it finds a semicolon in the input stream.

If the variable in the GET DATA statement is an aggregate, your input stream may specify any of the aggregate's scalar elements. Consider, for example, the effect of the two statements

```
DECLARE AR(100) FLOAT INITIAL((100)0);  
GET DATA(AR);
```

Because of the INITIAL option in the DECLARE statement, PL/I initializes all elements of the array to zeros. Now suppose that, when executing the GET DATA statement, PL/I finds the following in the input stream:

```
AR(5) = 18.23E5, AR(1) = 62, AR(50) = 42.7;
```

Then PL/I changes the values of the three array elements shown to the values shown, and all other elements of the array are left unchanged at the value zero. This example illustrates the general rule that the subscript used in the subscript list of the input stream corresponding to a GET DATA statement must always be an integer constant, possibly with a sign.

You may also use the GET DATA statement with no list of variables, as in the syntax

```
GET DATA;
```

In this case, your input stream may contain assignments for any variable in your program. (More precisely, the assignments may be for any variable whose declaration scope contains the GET statement.)

The GET EDIT Statement

Like PUT EDIT, the GET EDIT statement permits you to specify a format along with each data item to be transmitted. However, since GET LIST permits the input stream to contain free-format data input anyway, the main advantage that GET EDIT has over GET LIST is to permit you to use a format list to specify both the number of characters in the input stream to be used for each data item, and the number of spaces between data items.

Since GET EDIT is used more to control spacing than to specify the actual format of the input stream characters, you seldom use GET EDIT for input from your terminal. Instead, GET EDIT is used mostly for input from card, disk, or tape records that have been formatted in some particular way:

- Punched cards are often prepared offline in various commercial applications. Since space is at a premium on these cards, numbers are often punched in consecutive columns with no blanks at all between them. For example, the first few columns of a punched card might be

```
18265387025...
```

where 18265 is an employee identification number, 387 stands for 38.7 regular hours worked, and 025 stands for 2.5 overtime hours worked. You could not use GET LIST to read this input card, since no blanks or commas separate the fields. But you can use GET EDIT, where the format items specify the number of characters to be read for each data value. Furthermore, GET EDIT automatically inserts the implied decimal point, interpreting the input stream characters 387 as 38.7.

- You may create a disk file one day using PUT EDIT, and then on another day use that file as input. In that case, use GET EDIT with the same format items that you used to create the file.
- You may have a magnetic tape file prepared at a different installation or even on a different kind of computer with formatted character records.

As an example, look at the punched card example given above. That example shows the first 11 columns of a card, containing three data fields. The following is a GET EDIT statement that reads those three fields:

```
GET EDIT(IDENT, REG, OVT) (F(5), F(3,1), F(3,1));
```

This statement specifies input for three variables, IDENT, REG, and OVT. Input for the first variable is done in the format F(5), and for each of the last two in the format F(3,1). When PL/I inputs a value in the F(5) format, it takes five characters from the input stream. These five characters must be a DECIMAL FIXED constant, possibly with a sign. PL/I converts the characters to a numeric value and then assigns that numeric value to the variable, in this case the variable IDENT. PL/I inputs REG and OVT according to the format F(3,1). This format specifies input of three characters, with an implied decimal point preceding the last digit among those three characters. Therefore, if the three characters are 387, the value 38.7 is assigned to REG. Similarly, PL/I takes the input characters 025 and assigns them to OVT as 2.5.

Data Format Items: Later in this chapter, in the section entitled DETAILED SPECIFICATIONS FOR THE GET STATEMENT, precise specifications for each of the format items for GET EDIT are given. The most commonly used format items are

- F(w): PL/I takes w characters from the input stream. These w characters must form a decimal number, possibly with a sign, possibly with a decimal point, and possibly with leading or trailing blanks.
- F(w,d): PL/I takes w characters from the input stream. These w characters must be in the same format as has been described for F(w). The difference is that if the input characters do not

contain a decimal point, PL/I moves the decimal point in the input value to the left d places. (The decimal point is moved to the right if d is negative.)

- A(w): PL/I takes w characters from the input stream, and uses them as a CHARACTER stream of length w.
- A: PL/I takes characters from the input stream, stopping at the end of the current input line or record. PL/I puts these characters together to form a CHARACTER string value.

Note

This format item is not valid in ANS PL/I and will not be valid in other implementations of PL/I. ANS PL/I requires that the A format item for GET LIST specify w.

Table 11-2 illustrates the above format items. Additional illustrations of these format items are in the section DETAILED SPECIFICATIONS FOR THE GET STATEMENT.

Table 11-2
Common GET EDIT Format Items

Format	Input Stream	Input Value
F(5)	12345	12345
F(5)	123bb	123
F(5)	bb123	123
F(5,2)	12345	123.45
F(5,2)	123bb	1.23
F(5,2)	bb123	1.23
A(7)	ABCDEFG	'ABCDEFG'

The rules that PL/I follows for matching data variables with format items are the same as for the PUT EDIT statement. For example, suppose a punched card contains four-digit numbers, with 20 numbers per card. Then, the following statements can be used to cause PL/I to read the entire card as input and store all twenty data values into the array AR:

```

DECLARE AR(20);
...
GET EDIT(AR) (F(4));
    
```

Control Format Items: Like the PUT EDIT statement, GET EDIT also has a set of control format items that you may use to control the spacing between data items. The most commonly used are

- X(w): PL/I takes w characters from the input stream and ignores them.
- COLUMN(n): PL/I skips enough characters from the input stream so that the next input value comes from column n of the current input record. If input has already proceeded past column n in the current record, PL/I skips to the next input record, and then removes (n - 1) additional characters from the new input record, ignoring them.
- SKIP or SKIP(1): PL/I skips to the beginning of the next input record, ignoring all characters taken from the input stream to do this.
- SKIP(n), where n > 1: PL/I skips to the beginning of the next input record, and then skips (n - 1) additional records.

Notice that the SKIP format item is handled exactly the same as the SKIP option of the GET statement.

The following is an example of a GET EDIT statement, using the X format item:

```
GET EDIT(A, B) (F(5), X(3), F(3));
```

Suppose that, when PL/I executes this GET EDIT statement, it finds the following input stream:

```
12345QRS789
```

PL/I inputs the value of A using the format item F(5). This means that PL/I takes the characters 12345 from the input stream and assigns that value to the variable A. In performing input for the variable B, PL/I first encounters the control format item X(3), as a result of which it skips over the input stream characters QRS. Then, PL/I uses F(3) to input the characters 789, and to assign the value 789 to B.

The GET STRING Statement

Normally, PL/I executes the GET statement by taking characters from the input stream, usually from your terminal or from an external file or other device. By means of GET STRING, however, you can specify that PL/I is to take the input stream from a CHARACTER string expression of your choosing, rather than from an I/O device. The syntax is

```
GET STRING(expression) other-options;
```

When PL/I executes this statement, it evaluates the expression, converts it to a CHARACTER string value, if a conversion is necessary, and uses the characters in that CHARACTER string value as the input stream for the GET statement. An example is

```
DECLARE C CHARACTER(100) VARYING;
...
C = '123456';
GET STRING(C) EDIT(X, Y) (F(3), X(1), F(2));
```

In this GET statement, the input stream is the characters in the value of C, or 123456. When PL/I executes this statement, therefore, it assigns to X the value 123, skips over the character 4 as a result of the control format item X(1), and then uses the stream characters 56 to assign the value 56 to the variable Y, according to the format item F(2).

We may also use GET LIST with the STRING option, as in the following example, which concatenates a comma to indicate the end of one item.

```
C = '1234';
GET STRING(C||',') LIST(Z);
```

In this example, the string expression is C||',', which means that the input stream is 1234,. Therefore, Z is set equal to the value 1234, when PL/I executes the GET statement.

This last example illustrates the following two important points about using GET STRING:

- The string value used in the STRING option of GET may be any expression. (This is not true for PUT STRING, where the string value must be a variable reference.)
- If you are doing GET LIST with the string value, the usual rules about GET LIST terminators still apply. That is, each data value in the input stream must end with a blank or a comma, just like input from a file or device.

You may use the `STRING` option with `GET EDIT`, `GET LIST`, or `GET DATA`. The following options and format items are illegal with the `STRING` option: `SKIP`, `COLUMN`, and `FILE`.

Notice that the `GET STRING` statement is not an input/output statement, because no data is actually transmitted from an external device. Instead, it is a purely computational statement, since it manipulates data only within your program.

PUT AND GET TO FILES AND DEVICES

Usually you use the `PUT` statement to print data on your terminal, and you use the `GET` statement to take input values from the keyboard of your terminal. By following some special conventions, you may use `GET` and `PUT` to access all files and devices, such as those on disk or tape.

In any `PUT` or `GET` statement, use the `FILE` option to specify an identifier with the `FILE` data type. This `FILE` identifier can be associated with a file or device. For example, the statement

```
PUT FILE(F) LIST(X, Y);
```

specifies the `FILE` identifier `F`. The output from this `PUT` statement is transmitted to whatever device has been associated with this `FILE` identifier.

In fact, PL/I rules require that each `PUT` and `GET` statement must have either a `STRING` option or `FILE` option. If your program contains a `PUT` or `GET` statement with neither of these options, PL/I supplies a default `FILE` option. For a `GET` statement, the default `FILE` option is `FILE(SYSIN)`; for `PUT`, it is `FILE(SYSPRINT)`. Therefore, the statements

```
GET LIST(A, B);  
PUT LIST(A, B);
```

are entirely equivalent to the following statements:

```
GET FILE(SYSIN) LIST(A, B);  
PUT FILE(SYSPRINT) LIST(A, B);
```

Therefore, every `PUT` or `GET` statement that transmits data between your program and an external device has a `FILE` option, either by default or explicitly.

The FILE VARIABLE identifier appearing in the FILE option gives your input/output a kind of device independence. This means that the FILE identifier in your program can stand for one device and file one day, and a different device and file when you run the program the next day. Therefore, all your GET and PUT statements, as well as the other statements of your program, remain the same even though you may be doing input/output to different devices and files. The FILE VARIABLE is discussed in Chapter 12. The TITLE option discussed below also allows file independence.

DECLARE, OPEN, AND TITLE

Use the DECLARE statement to tell PL/I the name of your FILE identifier. For example, the statement

```
DECLARE F FILE;
```

specifies that F is such a FILE identifier. (In PL/I jargon, F is a file constant.) If this declaration appears in your program, you may use F in the FILE option of a PUT or GET statement. A DECLARE statement may also establish a file variable, which is discussed in Chapter 12.

The next step in using devices and files is the OPEN statement. The syntax of this statement is

```
OPEN FILE(file-name) [ INPUT ] [ OUTPUT ] [TITLE(character string)] ;
```

Use the OPEN statement to associate the specified file identifier with the device and file of your choice. After PL/I executes the OPEN statement, any GET or PUT statement using the same FILE identifier performs I/O to the device and file with which the OPEN statement associated the FILE identifier. This is explained more fully in the following paragraphs.

As the syntax shown above illustrates, in the OPEN statement you may specify either the input or output option. (A number of other options to the OPEN statement are discussed in the next section and the next chapter.)

If you specify the INPUT option, you will be using the file identifier in future GET statements; if you specify the OUTPUT option, you will be using the file identifier in PUT statements. If you do not specify either INPUT or OUTPUT, PL/I assumes INPUT.

You may use the TITLE option to specify which device and file you wish to use. The argument to the TITLE option may be any CHARACTER string. For example, the statement

```
OPEN FILE(F) INPUT TITLE('SQUIRREL');
```

would associate the file identifier F with the disk file called SQUIRREL, to be accessed for input. It would then be legal for your program to execute statements like

```
GET FILE(F) EDIT(X) (A(80));
```

This statement would input 80 characters from the disk file SQUIRREL, since the OPEN statement had associated the FILE identifier F with that disk file.

You may associate the file identifier with any device by using an appropriate CHARACTER string expression with the TITLE option. This is explained fully in a later section, Interpretation of the TITLE Option Argument.

Since the TITLE option may specify any PL/I expression, which may include arbitrary variables, it is possible for your program to determine at execution time the name of the device and file. Consider, for example, the following statements:

```
DECLARE F FILE;  
DECLARE C CHARACTER(100) VARYING;  
PUT LIST('TYPE NAME OF INPUT FILE:');  
GET LIST(C);  
OPEN FILE(F) INPUT TITLE(C);  
...  
GET FILE(F) LIST(C);
```

The first GET statement in this example inputs from the user's terminal the name of the file to be used as input later in the program. The name is assigned to the CHARACTER variable C, and then the OPEN statement uses that same variable in the TITLE option. The final GET statement of the example actually does input from that file.

You may, if you wish, omit the TITLE option from the OPEN statement. If you do, PL/I uses the FILE identifier name as the name of the disk file. For example, the statement

```
OPEN FILE(F);
```

associates the file identifier F with the disk file F to be accessed for input.

Refer to Chapter 12, the section FILE ATTRIBUTES, ATTRIBUTE MERGING, AND THE OPEN STATEMENT, for a discussion of required attributes, default attributes, and conflicting attributes implied by the OPEN statement.

Other Options of the OPEN Statement

The preceding section describes how you can use the most important options of the OPEN statement. Some other options of the OPEN statement are discussed below. The next chapter, on RECORD input/output, discusses the OPEN statement in more detail.

The syntax of the OPEN statement is

```
OPEN FILE(file-identifier) options;
```

where FILE(file-identifier) gives the name of the file identifier that is to be associated with an external file and device.

The following options may be specified in the OPEN statement:

- INPUT: this option specifies that the FILE identifier is to be used for input. Only GET statements may be used.
- OUTPUT: this option specifies that the FILE identifier is to be used for output. Only PUT statements may be used. If you specify neither the INPUT option nor the OUTPUT option, PL/I assumes INPUT as the default.
- STREAM: this option specifies STREAM I/O, the default, as opposed to RECORD I/O, which must be specified. Chapter 12 describes RECORD I/O.
- PRINT: this option is used only with OUTPUT files. If you use this option, you are specifying that the file is intended only to be printed, either on your terminal or on a line printer. You do not intend to use this file as input to another program. This option is explained more fully in a later section.
- LINESIZE(expression): this option may be used only for OUTPUT files. PL/I evaluates the expression and converts to an integer value, if necessary. This integer specifies the maximum number of characters in an output line or record. If a PUT statement outputs characters so that the line size is exceeded, PL/I automatically skips to a new line or record. The default is 120.

PL/I Reference Guide

- `PAGESIZE(expression)`: this option may only be used with `PRINT OUTPUT` files. PL/I evaluates the expression and converts it to an integer value. This integer specifies the maximum number of lines that may be printed on a single output page. After PL/I has printed that many lines, PL/I automatically skips to a new printing page. The default is 60.
- `TAB(expression, expression, ...)`: PL/I evaluates each expression and converts the value to an integer. Specify one or more of these expressions to determine the tab positions on the output line. Before doing any output, the `PUT LIST` statement moves to the next tab position on the output line.
- `TITLE(expression)`: this option is summarized above and is described more fully in the next section.

All arguments except `FILE` are optional. `INPUT` is the default. You may specify these options in any order in the `OPEN` statement. For example, the statement

```
OPEN TITLE('ANTLER') PRINT FILE(F) OUTPUT;
```

associates the file identifier `F` with the disk file `ANTLER` for use as `OUTPUT PRINT`.

Interpretation of the TITLE Option Argument

If your `OPEN` statement has a `TITLE` option, the argument of that option may be any PL/I expression. When PL/I executes the `OPEN` statement, PL/I evaluates this expression and converts it to a `CHARACTER` string. This `CHARACTER` string should have the following format:

```
'name [options]'
```

where the name is the name of either a disk file or a device, depending upon whether any options are specified. The name may be a filename with no options, in the format:

```
OPEN FILE(filename) TITLE('pathname [options]');
```

File pathnames specified in the `TITLE` option of a file statement can be up to 128 characters in length. Passwords can be included.

The options are

- **-DEVICE:** This option specifies that the name in the **TITLE** option character string is the name of a device, not the name of a disk file. The legal device names are

@TTY	the terminal
PTR	paper tape reader
PTP	paper tape punch
CR	card reader
SPR	serial printer
MT0-MT7	magnetic tape drives 0-7
PRO-PR1	line printers 0 and 1

- **-APPEND:** Use **-APPEND** with the **OUTPUT** option when you wish further output to go to the end of an existing file. If you do not use **-APPEND**, PL/I creates a new output file, deleting any existing file of the same name.
- **-ANSI:** Use **-ANSI** with the **-DEVICE** option specifying the terminal, to specify that ANSI standard PL/I conventions are followed with respect to line marks. That is, if **-ANSI** is specified, line marks are ignored in terminal input; otherwise, line marks act as terminators, exactly like commas and blanks.
- **-CITLASA:** See Chapter 12.
- **-FUNIT:** See Chapter 12.
- **-NOSIZE:** See Chapter 12.
- **-RECL:** See Chapter 12.
- **-FORMS:** See Appendix H.

If no **TITLE** option is given, the default is

```
OPEN FILE(f) TITLE('fname');
```

where fname is the declared name of the file constant referenced by f, and where f is a file variable, file-valued function, or file constant.

STREAM INPUT/OUTPUT SPECIFICATIONS

The rest of this chapter describes the concepts and specifications for STREAM input and output. This section describes standard STREAM I/O, as well as Prime's implementation of READ and WRITE statements with STREAM I/O.

Program Portability and Device Independence

The designers of the PL/I programming language were very concerned with the concept of program portability. To say that a program is portable usually means that if the program runs under one implementation of standard PL/I, it should run under other implementations of standard PL/I and get the same results.

Usually, discussions of program portability revolve around the difficulties of moving a program from one computer to another. The major problem in moving programs in this way is that different computers have different methods of representing numbers internally, and those differences can affect the way programs are written and the results that they produce.

However, there is another important aspect to the portability question: portability among input/output devices. Input/output to a tape can be very different from I/O to a disk. Even different disk drives can have very different input/output characteristics. Therefore, the problem facing the designers of the PL/I language was to design the input/output statements so that a program that is written to work with a given input/output device also works with other I/O devices, even when those devices are on different computers.

PL/I STREAM input/output solves these problems by the following means:

- All numeric data is transmitted to and from the input/output devices as machine independent decimal numbers in character form.
- All data on the input/output devices is considered to be organized as a stream of characters, split up into individual records.

Before describing these points in more detail, it is important to note that standard STREAM input/output statements (GET and PUT) afford program portability, but they are limiting because file access is sequential, from beginning to end (no random positioning).

The STREAM Input/Output Concept

When you use the PUT and GET statements, you are using PL/I STREAM input/output. STREAM I/O is designed to provide maximum program portability and device independence, concepts described above.

Note

Prime also supports READ and WRITE with STREAM I/O. However, because this is an extension to ANS PL/I, you can run programs using READ and WRITE with STREAM I/O only on Prime computers, thereby losing portability. READ and WRITE are presented later in this chapter.

An external collection of data is known as a data set and is informally called a file, the term used in this manual. PL/I STREAM input/output treats a file as a stream of characters organized into individual records. Some examples are the following:

- A file on punched cards is a stream of characters, where each record (each punched card) contains exactly 80 characters. A deck of 100 punched cards would consist of a stream of 8000 characters organized into 100 80-character records.
- A file on a line printer is printed in lines of characters. Each line is considered to be a record. (For more information about files directed to the line printer, see the PRINT attribute described in the next section.)
- When you type data to a PL/I program from the keyboard of your terminal, each record ends when you press the return key.
- Files on tape and disk devices may, in general, contain either binary or character data. However, a file to be accessed by means of STREAM input/output statements may contain only character data. (You may use RECORD input/output statements to access any kind of data.)

Although this way of looking at input/output devices is somewhat restricted, it does have the advantage of treating all I/O devices in the same way, thereby making programs portable.

STREAM input/output also helps with problems caused by data values having different internal representations on different computers. For example, suppose X is a FLOAT variable, and your program executes the statement

```
X = LOG10(2);
```

PL/I Reference Guide

PL/I computes the value of X and stores that value in the computer's memory in a special FLOAT format unique to that machine. However, if you execute the statement

```
PUT LIST(X);
```

PL/I converts that special internal format to a CHARACTER format, and prints something like

```
3.01030E-01
```

This CHARACTER representation of the value of X is the same for all computers, differing only in the number of significant digits.

Similarly, when your program executes a GET LIST statement, PL/I expects the input stream to contain a number in decimal digit CHARACTER format. PL/I converts these characters to the correct internal format for that computer.

This conversion of numeric values to and from a CHARACTER format is an important feature of STREAM input/output. No such conversions are done for RECORD input/output.

PUT and GET statements have a certain symmetry within PL/I STREAM input/output. The idea is that if you create a file using PUT statements, you may use that file as input to another PL/I program on the same or on a different computer, provided that you use the same sorts of options and format items in the GET statement that you used in the PUT statements when you created the file. For example, if you create the file with statements like

```
PUT SKIP EDIT(R, S, T)(X(5), F(20,2), COL(45), F(7,2), E(20,3));  
PUT LIST(U);
```

you should be able to read that same file later with the following statements:

```
GET SKIP EDIT(R, S, T)(X(5), F(20,2), COL(45), F(7,2), E(20,3));  
GET LIST(U);
```

Furthermore, you should be able to do that even if the second program is executing on a different computer. This might happen if one program creates a tape file on one computer and a second program on a different computer reads the same tape file.

Although the symmetry between PUT and GET is not perfect, it is followed very closely, and is an objective in the design of STREAM input/output.

PRINT Files

One important situation where the symmetry between PUT and GET operations breaks down is in PRINT files, that is, files that you open with the PRINT attribute. The concept is as follows: Some output files are intended to be printed only, on either your terminal or on a line printer. These files are not intended to be used as input later with GET statements, and so it is possible to relax some of the format rules for output for such files.

PL/I automatically gives to SYSPRINT, the default PUT file, the PRINT attribute. You may use the DECLARE or OPEN statement to give any other STREAM OUTPUT file the PRINT attribute.

When your program executes a PUT statement to a PRINT file, PL/I does certain things that make sense for a file that is to be printed but not used as later input. The two major differences between PRINT and non-PRINT files are described in the following paragraphs.

The more important difference has to do with paging of the output file. That is, if the file is intended to be printed, the records of a PRINT file are called lines, and these lines are organized into printer pages. This has the following implications:

- The PAGE and LINE options of the PUT statement are legal. Also, the PAGE and LINE format items of PUT EDIT are legal. PAGE and LINE are not legal for non-PRINT files.
- The PAGENO and LINENO built-in functions are legal only when the argument of the built-in function is a FILE with the PRINT attribute.
- The ENDPAGE condition can be raised only for SYSPRINT or other files with the PRINT attribute.

The other difference, not having to do with pages, is the following. When your program executes a PUT LIST statement for a data value that is either CHARACTER or pictured-character, PL/I encloses the output value in apostrophes if the file does not have the PRINT attribute; there are no apostrophes if the file has the PRINT attribute. For example, the statement

```
PUT LIST('THE ANSWER IS');
```

produces the following output to PRINT files:

```
THE ANSWER IS
```

and the following output to non-PRINT files:

```
'THE ANSWER IS'
```

STREAM File Information Pointers and Values

Standard PL/I looks at a STREAM file as a stream of characters organized into records. If the STREAM file is OUTPUT PRINT, the records are called lines, and the lines are organized into pages.

When your program is doing input from or output to a STREAM file, PL/I must keep track of several pointers and values related to the operations being performed on that file. To help you understand the STREAM input/output concept, some of these are listed here.

PL/I maintains a pointer to the last character read from or written into the file. This pointer is updated after every GET or PUT operation to the file. In the case of output, the pointer indicates the end of the file, or at least at the end of that portion of the file that has already been written.

PL/I also maintains the following values associated with the file operations:

- PL/I remembers the current position in the current record. PL/I needs this value in order to handle the COLUMN and TAB format items and to determine the position of the next tab stop for PUT LIST. Also, PL/I needs to know this value in order to know when a record has been filled.
- For PRINT files, PL/I remembers the current page number, which the user can obtain by calling the PAGENO built-in function. (The user can change this value by using the PAGENO pseudovisible.)
- For PRINT files, PL/I remembers the current line number on the printer page. It needs this value for use with the LINE option and LINE format item, as well as the LINENO built-in function. Also, PL/I must compare the line number with the value specified in the PAGESIZE option of the OPEN statement in order to determine when to raise the ENDPAGE condition.

In addition to the ANS handling of STREAM I/O shown above, Prime offers a way to use variable-length, unformatted STREAM I/O via the READ and WRITE statements. With READ and WRITE, Prime PL/I updates only the

pointer to the next line, not the pointer to the next character. READ and WRITE statements treat each line separately, one at a time.

The next section describes these statements and their use.

READ and WRITE With STREAM I/O

The use of READ and WRITE statements on STREAM files is an easy, fast, and efficient method of processing variable-length lines that need no formatting. It is the only way to process unformatted variable-length STREAM I/O one line at a time.

However, using READ and WRITE statements with STREAM files is not part of standard PL/I; therefore you can run programs that contain these statements only on Prime computers. Also, although programs using READ and WRITE run somewhat faster, for edited I/O you must use PUT and GET. Do not use READ and WRITE statements with PUT and GET statements on the same file.

Note

To process variable-length formatted STREAM I/O, you can use a GET statement with an A format that contains no field width. However, this format item is not valid in ANS PL/I and will not be valid in other implementations of PL/I. The section The A Input Data Format Item later in this chapter explains how to use this Prime implementation.

Only two options are available for READ and WRITE statements, but both options are mandatory. Each READ statement requires the FILE and INTO options, and each WRITE statement requires the FILE and FROM options. Each INTO and FROM option must specify a scalar varying character string variable.

A READ statement reads the next complete input line from the file variable or constant specified by the FILE option, and assigns it to the varying character string specified by the INTO option. A line is defined as everything up to but excluding a new-line character.

A WRITE statement writes any partial string currently in the output buffer to the output file. The statement then forces a line break so that the current value of the varying string specified by the FROM option begins on a new line.

The table below compares use of READ/WRITE and GET/PUT statements with STREAM I/O.

Table 11-3
READ/WRITE vs GET/PUT Statements

READ/WRITE	GET/PUT
Unformatted I/O	Formatted or unformatted I/O
File or device I/O	Terminal, file, or device I/O
Variable-length strings	Fixed-length strings only
One line for each operation	One field for each operation
Each string handled separately	Strings grouped into lines and pages
Character I/O only	Character I/O only
Required options FILE and INTO (READ) FILE and FROM (WRITE)	Required options SKIP, LIST, DATA, or EDIT (GET) SKIP, PAGE, LINE, LIST, DATA, or EDIT (PUT)
Valid statements ON ENDFILE ON ENDPAGE	Valid statements ON ENDFILE ON ENDPAGE FORMAT

The following sample program demonstrates use of the READ statement with STREAM I/O.

```

COUNT_LINES: PROCEDURE OPTIONS(MAIN);

/* This is a simple program to illustrate the use of nonstandard
   READ statements. */

DECLARE      F      INPUT FILE STREAM,
             SYSIN  INPUT FILE STREAM,
             SYSPRINT OUTPUT FILE STREAM,
             LENGTH BUILTIN,
             NAME   CHAR(256) VARYING,
             #LINES BIN INIT(0),
             #CHARS BIN INIT(0),
             S      CHAR(256) VARYING;

PUT FILE(SYSPRINT) LIST('INPUT FILE NAME: ');
    
```

STREAM INPUT/OUTPUT

```

GET FILE(SYSIN) EDIT(NAME) (A); /* Note: READ won't work here.
                                Nonstandard READ is for files
                                only, never for terminal input.*
/
OPEN FILE(F) TITLE(NAME);
ON ENDFILE(F) GOTO FINISH;      /* ENDFILE works even with
                                nonstandard READ. */

READ_LOOP: READ FILE(F) INTO(S); /* This is it, the nonstandard
    #LINES = #LINES + 1;          READ. You could also use GET
    #CHARS = #CHARS + LENGTH(S); FILE(F) EDIT(S) (A); here. */
GOTO READ_LOOP;

FINISH: CLOSE FILE(F);
PUT SKIP FILE(SYSPRINT);
PUT FILE(SYSPRINT)
    EDIT('FILE ', NAME, ' HAS ', #LINES, ' LINES AND ', #CHARS,
        ' CHARACTERS.')
        ((3)A, F(5), A, F(5), A);
PUT SKIP FILE(SYSPRINT);
END COUNT_LINES;

```

The following example demonstrates the use of READ and WRITE with STREAM I/O to copy files.

```

NONSTD_COPY: PROCEDURE OPTIONS(MAIN);

/* This sample program uses nonstandard READ and WRITE to copy files.
   Files are assumed to be text files. "binary" files cannot be
   copied in this way. */

DECLARE (SYSIN, SYSPRINT) FILE;
DECLARE (FROM, TO) FILE;
DECLARE LINE CHAR(1024)VARYING;

PUT FILE(SYSPRINT) EDIT('FILE TO COPY FROM: ') (A);
GET FILE(SYSIN) LIST(LINE);
OPEN FILE(FROM) TITLE(LINE) INPUT STREAM;

PUT FILE(SYSPRINT) EDIT('FILE TO COPY TO: ') (A);
GET FILE(SYSIN) LIST(LINE);
OPEN FILE(TO) TITLE(LINE) OUTPUT STREAM;

ON ENDFILE(FROM) GOTO DONE;

READING: READ FILE(FROM) INTO(LINE);
WRITE FILE(TO) FROM(LINE);
GOTO READING;

```

```
DONE: CLOSE FILE(FROM);
      CLOSE FILE(TO);

      PUT FILE(SYSPRINT) LIST('DONE!');
      PUT FILE(SYSPRINT) SKIP;

END NONSTD_COPY;
```

Use of ON ENDFILE

Once input for the GET statement comes from tape or disk files, you need a way to specify in your program what action your program should take when the GET statement fails because of end of file. Use the ON ENDFILE statement to specify this action.

Note

You also can use the ON ENDFILE statement for input with nonstandard READ statements. Refer to READ and WRITE With STREAM I/O earlier in this chapter for more information about nonstandard READ statements.

Consider, for example, the next sample program segment. This program segment inputs data values from a file, using the FILE identifier TAPEIN, and prints those values. In that example, the ON statement specifies that when end of file is reached on the file TAPEIN, PL/I is to transfer control to ENDTAPE. Notice that the program segment contains an infinite loop, since the condition specified in the WHILE clause of the DO statement is always true. This infinite loop actually terminates when the GET statement fails because of end of file. At that point, because of the ON ENDFILE statement, control transfers to the label ENDTAPE, and execution continues with the statement following the END statement of the DO loop.

```
      DECLARE TAPEIN FILE;
      OPEN FILE(TAPEIN) INPUT
        TITLE('MT2 -DEVICE');
      ON ENDFILE(TAPEIN) GO TO ENDTAPE;
      DO WHILE('1'B);
        GET FILE(TAPEIN) LIST(X);
        PUT LIST(X);
      END;
ENDTAPE: . . .
```

The next example illustrates another method for accomplishing the same thing, using the ON statement in a slightly different way. The ON ENDFILE statement specifies that when end of file is reached, the variable FLAG is to be set equal to one. In this case, the DO loop terminates as soon as end of file occurs.

```

DECLARE TAPEIN FILE;
FLAG = 0;
ON ENDFILE(TAPEIN) FLAG = 1;
GET FILE(TAPEIN) LIST(X);
  DO WHILE(FLAG = 0);
    PUT LIST(X);
  GET FILE(TAPEIN) LIST(X);
  END;

```

The syntax of the ON ENDFILE statement is

```
ON ENDFILE(file-identifier) on-unit;
```

where the on-unit can be one of the following:

- A GO TO statement
- An assignment statement
- Many other kinds of statements, but not an IF statement
- A group of statements, beginning with a BEGIN statement and ending with an END statement

The full rules for the syntax of the on-unit are given in Chapter 13, CONDITION HANDLING.

When your program executes the ON statement, an on-unit is said to become established for the ENDFILE(file-identifier) condition. If a GET statement fails because of end of file, PL/I automatically executes the statement or group of statements in the established on-unit. If the on-unit is a GO TO statement, the control passes to the specified target label. If there is no GO TO statement in the on-unit, then, after PL/I has finished executing the on-unit, PL/I transfers control to the statement following the GET statement that failed because of end of file. Chapter 13 describes this more fully.

If end of file occurs in the middle of a data item in the input stream, rather than between data items in the input stream, PL/I raises the ERROR condition rather than the ENDFILE condition.

Use of ON ENDPAGE

If you have opened your file with the PRINT attribute, you may use the ENDPAGE condition to control what happens after PL/I has completed each page of output. You can use the ENDPAGE condition with standard PUT statements as well as with nonstandard WRITE statements. Refer to the READ and WRITE with STREAM I/O section earlier in this chapter for more information about nonstandard WRITE statements.

Typically, use ON ENDPAGE to specify that a heading is to be printed at the top of each page of your output. For example, the following is a program segment example that prints a title and a page number at the top of each page:

```
P = 0;
ON ENDPAGE(SYSPRINT) BEGIN;
  P = P + 1;
  PUT PAGE EDIT('SUMMARY', 'PAGE', P) (A, COL(60), A, F(4));
END;
```

This example uses the variable P as a page number counter. The ON statement specifies the following: whenever a PUT statement reaches the bottom on the printer page, execute all the statements between the BEGIN and END statements, and continue with the PUT statement that caused output to reach the bottom of the printer page. This means that each time your PUT statement reaches the bottom of the PUT printer page, the statement P = P + 1 increases the page number counter by 1, and the PUT statement prints a heading something like

SUMMARY

PAGE 2

at the top of the next page.

Notice that PL/I raises the ENDPAGE condition whenever your printing reaches the bottom of the printer page. This means that there is no heading on the first page of output, unless you do something special. The easiest thing to do is to use the SIGNAL statement to raise the ENDPAGE condition artificially before you perform any other output. That is, before executing any PUT statement, execute the statement

```
SIGNAL ENDPAGE(SYSPRINT);
```

This statement artificially raises the ENDPAGE condition, just as if output had really reached the end of the printer page. The result is that the header line

SUMMARY

PAGE 1

is printed at the top of the first page of output.

The complete program example below illustrates all these concepts. This program prints out a table of the square roots of every number from 1 to 1000, with a heading at the top of each printer page. Note that

- The ON statement near the beginning of the program specifies what action PL/I should take when printing reaches the bottom of the printer page.
- The variable P keeps track of the page number.
- The SIGNAL statement artificially raises the ENDPAGE condition before any further output occurs, so that there is a heading on the first page of output.
- The DO group prints out the table without worrying about page headings. Whenever a PUT statement reaches the bottom of a page, PL/I automatically raises the ENDPAGE condition, with the result that a page heading is printed at the top of the next page.

Notice that the on-unit could use the PAGENO built-in function to determine the page number, rather than having a variable P.

```

SQRTAB: PROC OPTIONS(MAIN);
        P = 0;
        ON ENDPAGE(SYSPRINT) BEGIN;
            P = P + 1;
            PUT PAGE EDIT('X', 'SQRT(X)', 'PAGE', P)
                (A, COL(10), A, COL(60), A, F(4));
            END;
        SIGNAL ENDPAGE(SYSPRINT);
        DO X = 1 TO 1000;
            PUT SKIP EDIT(X, SQRT(X))(F(4), COL(10), F(10,3));
            END;
        END SQRTAB;

```

See Chapter 13, *CONDITION HANDLING*, for a full discussion of ON conditions. The next paragraphs contain some specific rules for the ENDPAGE condition.

When your program opens a PRINT file, the number of lines on a page is determined either by a system default value of 60, or by the PAGESIZE option of the OPEN statement. When your program executes a PUT statement to a PRINT file, PL/I keeps track of the line number on the printer page. Whenever the line number is incremented so that it equals the value of the page size plus one, PL/I raises the ENDPAGE condition. The standard system action for this condition is to execute a PUT PAGE operation to the output file.

If your program has established an ENDPAGE on-unit for the PRINT file, PL/I performs no automatic PUT PAGE. This means that your on-unit must do this explicitly, or else there will be two results:

PL/I Reference Guide

- If your on-unit does not do a PUT PAGE, the printer is not directed to skip to a new page. Depending on the characteristics of the line printer and on the value of the page size, this may mean that the following output does not appear at the top of the next page.
- Since PL/I resets the line number back to 1 only when your program executes an explicit PUT PAGE operation, PL/I continues incrementing the line number indefinitely. Thus ENDPAGE is never raised again unless you execute an explicit PUT PAGE statement.

ESTABLISHING DATA ITEMS

The parenthesized list immediately following the LIST, DATA, or EDIT keyword in a PUT or GET statement is called the list of data items. This section discusses how PL/I identifies individual data items, especially when aggregates are involved.

Aggregates in Data Item Lists

If the list following the DATA, LIST, or EDIT keyword contains an aggregate variable or expression, all scalar elements of the aggregate are considered to be items in the data item list. For example, the statement

```
DECLARE AR(100);  
PUT LIST(AR);
```

specifies the aggregate AR in the output list. As a result, there are 100 data items to be printed.

Implied DO Loops in Data Item Lists

In place of a variable or expression, you may use the following syntax in the data item list:

```
(variable-list DO do-options)
```

or

```
(expression-list DO do-options)
```

where the phrase DO do-options can be any legal DO statement (without the semicolon). Notice that this syntax includes a set of parentheses.

For example, in

```
DECLARE AR(100);
...
GET LIST((AR(I) DO I = 1 TO 10));
```

the GET statement inputs the values of AR(1), AR(2), ..., AR(10). On the other hand, the statement

```
PUT DATA((AR(K) DO K = 1 REPEAT (2*K) WHILE(K <= 64)));
```

prints, in PUT DATA format, the values of AR(1), AR(2), AR(4), AR(8), AR(16), AR(32), and AR(64). The following statement prints the array AR in a two-column table, the first column printing the values 1 through 100, and the second column the values of AR(1), AR(2), ..., AR(100):

```
PUT EDIT('I', 'AR(I)', (I, AR(I) DO I = 1 TO 100))
(PAGE, A(10), A, 100(SKIP, F(3), F(15,2)));
```

With two-dimensional arrays, you may wish to nest the DO clauses. In the statements

```
DECLARE ARD(3,100);
...
PUT LIST(((ARD(I,J) DO I = 1 TO 3)
DO J = 100 TO 1 BY -1));
```

the PUT statement prints the array ARD in the order ARD(100,1), ARD(100,2), ARD(100,3), ARD(99,1), ..., ARD(1,3).

MATCHING DATA VALUES TO FORMAT ITEMS

When you execute a PUT or GET statement with the EDIT option, you must specify both a list of data items and a list of format items. PL/I must match each data item to a corresponding data format item.

The general rule is that PL/I alternates between the data item list and the format list. PL/I determines a data item according to the rules given in the preceding section, then searches for the next format item. This process stops when the data items are used up.

PL/I Reference Guide

For example, in the statement

```
PUT EDIT(A, B) (F(2), F(3), F(4));
```

the value of A is printed in the F(2) format, and B in the F(3) format. After B is printed, the PUT statement is finished, and the F(4) format item is ignored.

On the other hand, if the format list is used up before all the data items have been processed, the format list wraps around; that is, PL/I starts the format list over again from the beginning. Consider, for example, the following statement:

```
GET EDIT(A, B, C, D, E) (F(2), F(3));
```

This GET statement inputs A in the format F(2), and B in the format F(3). Then, since the format list is ended, PL/I starts it at the beginning and uses F(2) to input the value of C. This process continues, using F(3) for D, and F(2) for E.

Kinds of Format Items

There are three classes of format items:

- Data format items
- Control format items
- Remote format items

The general rule given above about searching for a format item is modified as follows: for each data item, PL/I searches the format list for the next data format item. If that search encounters some control or remote format items, appropriate action is taken, and the search for a data format item continues. For example, given

```
PUT EDIT(A, B) (F(2), LINE(20), X(5), F(8), SKIP, F(10));
```

PL/I prints the value of A using the F(2) data format item. To print B, PL/I must search for the next data format item in the format list. In searching, PL/I processes the control format items LINE(20) and X(5) by skipping to the appropriate line and then printing five blank characters. Finally, PL/I encounters the data format item F(8), and prints B in that format. Since that ends the data item list, execution of the PUT statement terminates, and the SKIP and F(10) format items are ignored.

The Remote Format Item and the FORMAT Statement

For convenience, PL/I permits you to put format items into a separate FORMAT statement, and then to refer to that FORMAT statement from the format list in the EDIT option of the PUT or GET statement. Do this by means of the remote format item, R.

Consider the following example:

```

GET SKIP EDIT(A, B, C)
      (F(2), R(FRM), COL(20), F(5));
      . . .
FRM:  FORMAT(X(2), F(4), SKIP);

```

In this example, the format list of the GET statement contains the remote format item R(FRM), which refers to the FORMAT statement with statement label FRM. The effect of the R format item is to reference the format items in the FORMAT statement as a kind of format list subroutine, with the result that PL/I uses the format items in the FORMAT statement as if they appeared in the format list of the EDIT option of the GET statement. When PL/I executes this statement, PL/I matches the data item A with the data format item F(2). Then, for the data item B, PL/I searches for a corresponding data format item, performing the following operations:

1. PL/I encounters the remote format item R(FRM) and goes to the format statement FRM.
2. There, PL/I encounters the control format item X(2), takes two characters from the input stream, and ignores them.
3. PL/I encounters the data format item F(4); the search is ended.

Next, PL/I must find the data format item corresponding to the data item C. It does this by continuing from the point where it left off in the FORMAT statement, as follows:

4. PL/I encounters the SKIP control format item, and so skips to the beginning of the next input record.
5. Because that is the end of the format list in the FORMAT statement, PL/I goes back to the point in the format list of the GET statement just after the remote format item R(FRM).
6. PL/I encounters the COLUMN(20) control format item, and so skips to column 20 in the current input record.
7. PL/I encounters the F(5) data format item; the search is ended.

PL/I Reference Guide

Because that is the end of the data list of the GET statement, the GET operation is complete.

It is legal for one FORMAT statement to reference another by means of the R format item in the format list of a FORMAT statement.

Format Repetition Factors

Use a repetition factor in a format list to specify that a format item, or group of format items, is to be used more than once. The syntax is

```
n format-item
```

Consider, for example, the statement

```
PUT EDIT(A, B, C, D, E) (2 F(2), X(5), 3 F(4));
```

This statement has the same effect as

```
PUT EDIT(A, B, C, D, E) (F(2), F(2), X(5), F(4), F(4), F(4));
```

In the first of these two PUT statements, the repetition factors 2 and 3 specify that the format items that follow are to be used that many times.

You may specify a repetition factor for an entire list of format items by using the syntax

```
n(format-list)
```

For example, the statement

```
GET EDIT(A, B, C, D, E) (F(2), 3(X(2), F(4)), F(8));
```

is the same as the statement

```
GET EDIT(A, B, C, D, E) (F(2), X(2), F(4),  
X(2), F(4), X(2), F(4), F(8));
```

where the format list (X(2), F(4)) is used three times, as specified by the repetition factor 3.

You may also use repetition factors within repeated format lists, giving the effect of nested repetitions. Consider the following statements:

```
DECLARE ARD(100,10);
...
PUT EDIT(ARD) (100(SKIP, 10 F(8)));
```

The PUT statement has a format list with nested repetition factors. It prints 100 lines of data, each line containing 10 data values.

Variables and Expressions in Format Lists

In the examples we have seen so far, the numeric values in format items have all been decimal constants. PL/I permits arbitrary variables and expressions in all such places. This permits you to write your program so that field widths and other numeric specifications can be decided when your program executes, rather than when your program is written.

In the program segment

```
IF X > 999 THEN W = 7; ELSE W = 3;
PUT EDIT(X) (F(W));
```

the IF statement tests the value of X to determine whether it can be printed out using three characters. If it can, the variable W is set equal to 3; otherwise, the variable is set equal to 7. The value of W is used in the F format item of the PUT EDIT statement and so the number of characters printed is determined at execution time rather than compile time.

A somewhat trickier example, using the mathematical built-in function LOG10, illustrates the use of an arbitrary expression with a format item:

```
PUT EDIT(X) (F(1 + TRUNC(LOG10(X))));
```

The expression $1 + \text{TRUNC}(\text{LOG}_{10}(X))$ can be shown mathematically to equal the number of decimal digits in the value of X, provided that X is a positive integer. Therefore, this PUT EDIT example prints the value of X with no leading blanks.

PL/I Reference Guide

You may also use variables and expressions in format list repetition factors, provided that you enclose the repetition factor in an extra set of parentheses. The syntax is either of the following two forms:

```
(expression) format-item
(expression)(format-list)
```

As an example, consider the statements

```
DECLARE AR(100);
...
PUT EDIT(AR) (SKIP, (N) F(5));
```

The PUT statement prints the array AR in a table with N columns and as many lines as are necessary.

If a repetition factor in a format list is a variable or expression whose value is 0, PL/I skips the entire format item or format list. Consider the following example:

```
IF ABS(X) >= 1E8 THEN SW = 0; ELSE SW = 1;
PUT EDIT(X) ((SW)F(9), E(9,2));
```

In this example, the variable SW is used as a switch to determine which of the format items, F(9) or E(9,2) is to be used to print the value of X. If X is small enough to be printed with nine characters, the IF statement sets SW = 1, so that F(9) is used; otherwise, X is printed in scientific notation, using E(9,2).

FORMAT Variables

PL/I considers the identifier in the statement label of a FORMAT statement to be a constant of the FORMAT data type. Since, therefore, PL/I has constants of the FORMAT data type, it also permits you to use the DECLARE statement to create a variable of the FORMAT data type. You may use that variable in an R remote format item, after assigning to it the value of an appropriate FORMAT constant.

In the program segment

```
DECLARE FV FORMAT VARIABLE;  
IF X >= 1E8 THEN FV = FRM2;  
    ELSE FV = FRM1;  
PUT EDIT(X) (R(FV));  
FRM1:  FORMAT(F(9));  
FRM2:  FORMAT(E(9,2));
```

the PUT EDIT format list contains a remote FORMAT variable FV. The IF statement sets the variable FV to equal one of the FORMAT constants FRM1 or FRM2. Thus, the value of X determines whether the format F(9) or E(9,2) is used to print it.

DETAILED SPECIFICATIONS FOR THE PUT STATEMENT

This section provides detailed specifications for each of the options of the PUT statement and for each of the format items of the PUT EDIT statement. All the options and format items are listed in alphabetical order. Notice that some keywords such as SKIP are both options and format items, and are handled the same way.

For each option and format item, the list gives its syntax and the rules that PL/I follows in executing it. For data format items, the rules specify what conversions are made and what the precise format of the output is. In all cases, the rules specify what characters are placed in the output stream. Note, however, that in the actual execution of your program, the rules specified by this section may be modified as follows:

- When the output stream reaches the end of an output record, or a print line, PL/I performs a PUT SKIP automatically.
- If the output reaches the end of a page for a PRINT file, PL/I signals the ENDPAGE condition. If your program has established an ENDPAGE on-unit, PL/I invokes that on-unit, even in the middle of the output described by one of the rules in this section. Upon normal return from the on-unit, PL/I continues doing output according to the rules in this section, continuing execution with the statement that caused the ENDPAGE condition to be raised.
- If ENDPAGE on-unit is not called for, PL/I continues output according to the rules in this section. Execution continues with the statement that caused the ENDPAGE condition to be raised.

The A Output Data Format Item

Purpose: Specifies the number of characters in the output.

Syntax: A or A(w), where w, if specified, is any expression. PL/I evaluates w and converts it to an integer. It is an error for w to be negative.

Rules: Let V1 be the value of the scalar data item that is being printed with the A format item.

1. PL/I converts V1 to the CHARACTER data type, to get a new value, V2.
2. If w is not specified, let $w = \text{LENGTH}(V2)$.
3. If w is specified, PL/I pads with blanks or truncates, if necessary, the string V2, so that its length is w characters.
4. PL/I places the characters of V2 into the output stream.

Examples: Table 11-3 displays various examples of the A format item. Line 1 illustrates A with no value for w. In that case, the number of characters that PL/I places in the output stream equals the length of the data value. Lines 2 through 4 show the effect of specifying w. In line 5, PL/I converts the FLOAT value 2.84E+02 to the CHARACTER value 'bb2.84E+02'. When PL/I prints this value in the A(12) format, two more blanks appear on the right.

Table 11-3
The A Output Format Item

Line #	Data Value	Format Items	Output Chars
1	'XYZ'	A	XYZ
2	'XYZ'	A(5)	XYZbb
3	'XYZ'	A(2)	XY
4	'XYZ'	A(0)	(none)
5	2.84E+02	A(12)	bb2.84E+02bb

The B, B1, B2, B3, and B4 Output Data Format Items

Purpose: Specify the radix for bit string output.

Syntax: B, B1, B2, B3, B4, B(w), B1(w), B2(w), B3(w), or B4(w), where w, if specified, is any PL/I expression. PL/I evaluates w, if specified, and converts it to an integer. It is an error for w to be negative.

Rules: Let V1 be the value of the scalar data item that is being printed with the bit format item. In the following paragraphs, let n be the radix factor of the format item. That is, the format item is Bn or Bn(w), where n equals 1, 2, 3, or 4, and where B is the same as B1.

1. PL/I converts V1 to the BIT data type, to get a new value, V2.
2. If w is not specified with the format item, let $w = \text{CEIL}(\text{length}(V2)/n)$. That is, if w is not specified with the format item, the default value of w is the precise number of characters needed to print the BIT string value V2 as a BIT string with the specified radix factor.
3. If w is specified, PL/I pads with 0-bits or truncates, if necessary, the BIT value, V2, so that it contains precisely n*w bits.
4. PL/I divides the BIT string V2 into w groups of bits, each containing n bits.
5. Using Table 11-4, PL/I converts each of the w groups of n bits into one of the characters 0-9 or A-F. The result of this is a CHARACTER string value V3 containing w characters.
6. PL/I places the characters of V3 into the output stream.

Table 11-4
 Characters Corresponding to B(n)

B or B1	Format Item			Corresponding Character
	B2	B3	B4	
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	A
			1011	B
			1100	C
			1101	D
			1110	E
			1111	F

Examples: Table 11-5 illustrates the various bit format items. Lines 1 through 5 illustrate the B and B1 format items; these are the same. In line 2, the BIT value '01101'B, which is being printed in the B(7) format, is padded with two 0-bits. In lines 3 and 4, PL/I must truncate.

Table 11-5
The B Output Format Item

Line #	Data Value	Format Items	Output Chars
1	'01101'B	B	01101
2	'01101'B	B(7)	0110100
3	'01101'B	B(3)	011
4	'01101'B	B(0)	(none)
5	'01101'B	B1	01101
6	'011011'B	B2	123
7	'011011'B	B2(4)	1230
8	'01101'B	B2	122
9	'011101001'B	B3	351
10	'011101001'B	B3(2)	35
11	'0111'B	B3	34
12	'01101'B	B3	32
13	'01011000'B	B4	58
14	'11011110'B	B4	DE
15	'000011'B	B4	0C
16	'01'B	B4(3)	400

Lines 6 through 8 of the same table illustrate the B2 format items. In each case, the data value is '011011'B, and PL/I breaks up the bits into three groups of two bits each, to get 01, 10, and 11. Looking at Table 11-4, in the B2 column, we see that these groups of bits correspond to the characters 1, 2, and 3, respectively, and so PL/I prints 123. Line 7 is similar, except that w equals 4 in the format item, and so PL/I pads the bit string, and prints 1230. Line 8 illustrates what happens when there is an odd number of bits in the BIT data value associated with the B2 format item. PL/I pads the BIT value '01101'B to get an even number of bits, so that the result is '011010'B. PL/I breaks up this string of bits into the groups 01, 10, and 10, corresponding to the characters 1, 2, and 3, respectively.

Lines 9 through 12 illustrate the B3 format items. In line 9, PL/I breaks up the BIT value '011101001'B into groups of three bits each, to get 011, 101, and 001. Using the first row, we see that these bit groups correspond to the characters 3, 5, and 1, respectively, and so PL/I prints 351. Line 10 is the same, except that, since w equals 2, PL/I must truncate the result to the two characters, 35. Lines 11 and 12 illustrate what happens when the number of bits in the data value is

not divisible by three. In line 11, PL/I extends the data values '0111'B to '011100'B, and in line 12, PL/I extends '01101'B to '011010'B.

Lines 13 through 16 illustrate that B4 works much the same as B3 and B2. In line 14, PL/I breaks up '11011110'B into groups of four bits each, to get 1101 and 1110, corresponding to the characters D and E, respectively, so that PL/I prints DE.

In line 16, since w equals 3 and n equals 4, PL/I extends the BIT value '01'B to $3*4$, or 12 bits, to get '0100000000'B, so that 400 is printed.

The C Output Data Format Item

Purpose: Provides special formats for complex numbers.

Syntax: C(form1) or C(form1, form2), where form1 and form2 are each one of the following data format items: F, E, or P. In the case of P, the picture specification must be pictured-numeric, rather than pictured-character.

Rules: Let V1 be the value of the scalar data item that is being printed with the C format item.

1. PL/I converts V1 to COMPLEX to get a new value, V2.
2. If form2 is not specified, let form2 = form1.
3. PL/I prints the real part of V2 with the form1 format item, and prints the imaginary part of V2 with the form2 format item.

Examples: Table 11-6 illustrates the C format item. In line 1, each of the values 4 and 3 is printed using the format item F(2). The result is b4b3. In line 2, the same pair of values is printed using F(2) and F(1) formats. Note that the result has no blank character between the 4 and the 3. In line 3, the REAL value 4 is converted to the COMPLEX value 4 + 0I, and the real and imaginary parts are printed using the format item F(2).

Table 11-6
The C Output Format Item

Line #	Data Value	Format Item	Output Chars
1	4+3I	C(F(2))	b4b3
2	4+3I	C(F(2), F(1))	b43
3	4	C(F(2))	b4b0

The COLUMN Output Control Format Item

Purpose: Puts output in the desired column.

Syntax: COLUMN(n), where n is any PL/I expression. PL/I evaluates n and converts it to an integer. It is an error for n to be negative. If n equals 0, or if n is greater than the line size for the output file, PL/I changes the value of n to 1.

Abbreviation: COL for COLUMN.

Rules:

1. If the current column number in the current print line or output record of the file is greater than the value of n, PL/I performs a PUT SKIP to the file.
2. PL/I prints a sufficient number of blank characters so that, when this operation has been completed, the next output character goes in column n. Specifically, if there are already k characters in the current output print line or record, where $k < n$, PL/I puts $(n - k - 1)$ blank characters into the output stream.

The DATA Option

Purpose: Prints data values with their names (good for debugging).

Syntax: DATA(list) or DATA.

The method for determining the individual data items from the list has been described in the section on The GET DATA Statement. The list may include aggregates and implied DO groups. The individual data items may not be arbitrary expressions, but must be variables, structure elements, or array elements with arbitrary expression subscripts. No data item may be BASED or have a noncomputational data type.

If you use DATA with no list, PL/I prints the values of all variables in your program that are accessible to the statement, including those that are declared in containing blocks.

Rules: PL/I prints the individual data items in the following format:

```
name = value      name = value      . . .      name = value;
```

It uses the the following rules:

1. Before each name, PL/I inserts a blank into the output stream. If output is going to a PRINT file, PL/I inserts enough blank characters to advance to the next tab stop.
2. If there is not enough room in the current record (print line) for the entire name, PL/I executes a PUT SKIP to the output file.
3. PL/I prints the name of the variable or aggregate element being printed, with all structure qualifiers and subscripts. If the item is an element of an array of structures, the subscripts are not interleaved; instead, the entire subscript list is printed in parentheses to the right of the entire qualified name. See the example below.
4. PL/I prints the value of the variable in the same format as it would for PUT LIST to a non-PRINT file. See the discussion of the LIST option for more details of this format.

After PL/I has printed all individual data values specified by the DATA option, PL/I prints a semicolon.

Examples:

```
DECLARE A FIXED DECIMAL(5), B CHARACTER(100) VARYING;  
...  
PUT DATA(A, B);
```

The PUT statement of this program segment prints the values of A and B in a format that appears something like the following:

```
A =          23          B = 'STRING VALUE';
```

Notice that the values are printed out as individual assignments, and the last such assignment ends with a semicolon.

The following example uses an array of structures:

```
DECLARE 1 S (100),  
        2 U FIXED DEC(5),  
        2 V(10) FIXED DEC(10);  
...  
PUT DATA(S);
```

In this example, the PUT statement prints out all 1100 elements of the array of structures S. All subscripts are printed to the right of the fully qualified name, and so are not interleaved within the individual components of the fully qualified name. For example, the value of S(23).U might be printed as something like

```
S.U(23) = -234
```

with the subscript (23) appearing to the right of the fully qualified name S.U, rather than being interleaved between the components of the name, as it would be if it had been printed in the format S(23).U. Similarly, PL/I would print the value of S(23).V(5) in a format something like

```
S.V(23, 5) = 45
```

where the two subscripts are combined into a single subscript list appearing to the right of the fully qualified name.

The E Output Data Format Item

Purpose: Provides scientific notation formats.

Syntax: E(w), E(w, d), or E(w, d, s), where w, d, and s are any PL/I expressions. PL/I evaluates each of w, d, and s that is specified and converts it to an integer.

Rules: Let V1 be the value of the scalar data item that is being printed with the E format item.

1. If neither d nor s is specified in the E format item, let s be the converted precision for conversion of V1 to REAL FLOAT DECIMAL, where the converted precision is defined in Chapter 6, and let $d = s - 1$.
2. If d is specified in the E format item, but s is not, let $s = d + 1$.
3. In any case, it is an error if $w < 0$, or $d < 0$, or $s < d$.
4. PL/I converts V1 to the data type REAL FLOAT DECIMAL(s), to get a new value, V2.
5. PL/I prints V2 in scientific notation according to the following rules: First, if $s = d$, PL/I prints V2 in the following format:

For $S \leq 6$,

$$\underbrace{b \dots b}_{w-7-d} - 0 \underbrace{. 9 \dots 9}_d E \pm 99$$

For $S > 6$,

$$\underbrace{b \dots b}_{w-9-d} - 0 \underbrace{. 9 \dots 9}_d E + 9999$$

Next, if $d = 0$, PL/I uses the format

For $S \leq 6$,

$$\underbrace{b \dots b}_{w-5-s} - \underbrace{9 \dots 9}_s E \pm 99$$

For $S > 6$,

$$\underbrace{b \dots b}_{w-7-s} - \underbrace{9 \dots 9}_s E + 9999$$

If neither of the above two cases hold, PL/I uses the following format:

For $S \leq 6$,

$$\underbrace{b \dots b}_{w-6-s} - \underbrace{9 \dots 9}_{s-d} . \underbrace{9 \dots 9}_d E + 99$$

For $S > 6$,

$$\underbrace{b \dots b}_{w-8-s} - \underbrace{9 \dots 9}_{s-d} . \underbrace{9 \dots 9}_d E + 9999$$

In each of these formats, 9 stands for a digit position, either in the mantissa or the characteristic portion of the output. The minus sign is replaced by a blank if the mantissa is nonnegative. The characteristic of the value is printed right after the E, and PL/I always prints a sign for the characteristic even if it is positive.

6. If w is too small to accommodate all the signs and digits required by the format described above, the format item is in error, and PL/I prints w question marks.

Examples: Table 11-7 illustrates the E format items. Lines 1 through 6 illustrate the format E(w , d). In this case, PL/I prints the data value with one digit before the decimal point, and d digits after the decimal point. If $d = 0$, as in line 6, PL/I prints no decimal points.

Table 11-7
The E Output Format Item

Line #	Data Value	Format Item	Output Chars
1	23	E(9,2)	b2.30E+01
2	23	E(8,2)	2.30E+01
3	-23	E(9,2)	-2.30E+01
4	-23	E(8,2)	(size error)
5	23	E(9,1)	bb2.3E+01
6	23	E(9,0)	bbbb2E+01
7	23	E(11,2,3)	bbb2.30E+01
8	23	E(11,2,2)	bbb0.23E+02
9	23	E(11,2,4)	bb23.00E+00
10	23	E(11,2,5)	b230.00E-01
11	23	E(11,2,6)	2300.00E-02
12	23	E(11,0,6)	b230000E-04
13	23	E(11)	bbbb2.3E+01
14	023.0	E(11)	bb2.300E+01

Lines 7 through 12 illustrate E(w, d, s). PL/I prints a total of s significant digits in the mantissa, with d of them after the decimal point. If d = s, as in line 8, PL/I prints an extra zero preceding the decimal point, since all significant digits follow the decimal points in that case. If d = 0, as in line 12, PL/I prints no decimal point.

Lines 13 and 14 illustrate the format E(w), where d and s are not specified. In this case, PL/I prints a number of digits in the mantissa(s) equal to the number of digits in the converted precision for the conversion of the data value to REAL FIXED DECIMAL. (See Chapter 6 for information on converted precision.) PL/I prints these s digits with one digit before the decimal point and (s - 1) digits after the decimal point. In line 13, the converted precision is 2, so E(11) is equivalent to E(11,1,2). In line 14, the converted precision is 4, and so E(11) is equivalent to E(11,3,4).

The EDIT Output Option

Purpose: Provides special formats for numeric items.

Syntax: EDIT(data-list) (format-list) or
EDIT(data-list) (format-list) (data-list) (format-list)...

The data-list can contain any PL/I expressions with computational data types. The individual scalar data items are matched up with format items in the manner that has been described in the section on The PUT EDIT Statement.

In your EDIT option, you may specify two or more data-list and format-list pairs. PL/I matches the data items in the first data-list to the format items in the first format list, the data items in the second data-list to the format items in the second format list, and so forth.

The F Output Data Format Item

Purpose: Provides special fixed decimal formats for numeric items.

Syntax: F(w) or F(w, d), or F(w, d, s), where each of w, d, and s, if specified, is any PL/I expression. PL/I evaluates each operand specified and converts it to an integer.

Rules: Let V1 be the value of the scalar data item that is being printed with the F format item.

1. If d is not specified in the F format item, let $d = 0$. If s is not specified, let $s = 0$.
2. It is an error for either w or d to be negative; if either is negative, it is treated as zero.
3. PL/I converts the value V1 to the data type REAL FIXED DECIMAL(p, d), where p is the number of digit positions available in the output field. The value of p is obtained by subtracting from w the number of character positions required for the decimal point and sign. PL/I multiplies the result of this conversion by 10^s , and rounds off the result to d digit positions after the decimal point. Let V2 be the result of these computations.
4. PL/I represents V2 as a string of characters, right adjusted in the field of length w. If $d > 0$, there is a decimal point in the representation, with d digits following the decimal point. If $V2 < 0$, there is a minus sign just before the first digit. All other character positions are leading blanks.

In the above representation, leading zeros are handled as follows: PL/I always puts at least one digit before the decimal point. (If there is no decimal point, there is always at least one digit.) This means that if the value being printed is less than 1 in absolute value, PL/I puts a leading zero just before the decimal point. All digits following the decimal point are present, even if they are all zeros.

5. PL/I puts the characters in this CHARACTER string representation into the output stream.
6. If there is any error in the format item, or if w is too small to accommodate the resulting CHARACTER string, PL/I puts w question marks into the output stream.

Examples: Table 11-8 illustrates the F format item.

Lines 1 through 6 illustrate F(w), where d and s are not specified. PL/I rounds the value to an integer, and right adjusts it in the output field.

Lines 5 through 9 illustrate F(w, d). PL/I prints d digits after the decimal point. Notice in lines 8 and 9 that there is a leading zero printed before the decimal point when the value being printed is less than 1 in magnitude.

Lines 10 through 14 illustrate F(w, d, s). The effect of s is to move the decimal point to the right s digit positions. (If s is negative, the decimal point moves to the left.) Thus, for example, in line 10, 23 printed in the format F(6,0,1) is printed as a value of 230. The other examples are similar.

Table 11-8
The F Output Format Item

Line #	Data Value	Format Item	Output Chars
1	23	F(6)	bbbb23
2	23.7	F(6)	bbbb24
3	-23.4	F(6)	bbb-23
4	0	F(6)	bbbbbb0
5	23	F(6,2)	b23.00
6	23.7	F(6,2)	b23.70
7	.03	F(6,2)	bb0.03
8	-.74	F(6,2)	b-0.74
9	-.748	F(6,2)	b-0.75
10	23	F(6,0,1)	bbb230
11	23	F(6,0,2)	bb2300
12	23	F(6,2,1)	230.00
13	23	F(6,2,-1)	bb2.30
14	23	F(6,2,-2)	bb0.23

The FILE Output Option

Purpose: Identifies the file to which output is sent.

Syntax: FILE(reference), where the reference is an expression whose data type is FILE. Usually the reference is to a FILE constant, as explained in the section on PUT and GET to files and devices, but it may also be a FILE variable or a FILE value returned by a user-defined function. FILE variables and user-defined functions are explained in Chapter 12, RECORD INPUT/OUTPUT.

The LINE Output Option and Control Format Item

Purpose: Puts output on the desired line of a page.

Syntax: LINE(n), where n is any PL/I expression. LINE may be an option of the PUT statement, or a control format item in a PUT EDIT format list. PL/I evaluates n and converts it to an integer. It is an error for n to be negative. The file to which the PUT operation is specified must have the PRINT attribute.

Rules: PL/I performs as many PUT SKIP operations as necessary to move to the beginning of line n on the current page of output.

If, however, printing has already passed the beginning of line n on the current page, PL/I performs as many PUT SKIP operations as necessary to move to the bottom of the current page, and then raises the ENDPAGE condition, for which the standard system action is to perform a PUT PAGE operation. PL/I does the same thing if the value of n is greater than the value of the page size (the number of lines on a page).

Examples:

```
PUT EDIT(X, Y, Z) (PAGE, F(10), SKIP, F(10), LINE(20), F(5));
```

This statement is equivalent to

```
PUT EDIT(X, Y, Z) (PAGE, F(10), SKIP, F(10), SKIP(18), F(5));
```

because the LINE(20) control format item comes in the middle of the second line of output on the current page.

If now your program executes

```
PUT LINE(5) LIST(X);
```

PL/I performs enough PUT SKIP operations to move to the bottom of the printer page, and then signals the ENDPAGE condition.

The LIST Output Option

Purpose: Provides character and bit string formats for output.

Syntax: LIST(data-list)

The data-list can contain any expressions, including aggregate expressions and implied DO groups. The individual data items are established as described in the section ESTABLISHING DATA ITEMS. Each expression must have a computational data type.

Rules: Before printing each scalar data item, PL/I inserts a blank character into the output stream. If output is through a PRINT file, PL/I inserts enough blank characters to advance to the next tab stop position.

Let V1 be the scalar data item being printed. PL/I converts V1 to CHARACTER, following the rules in Chapter 6 to obtain a new CHARACTER value, V2. Then PL/I proceeds as follows, depending upon the data type of V1:

1. If V1 is BIT, PL/I places the following in the output stream: an apostrophe, all the characters in V2, another apostrophe, and a B.
2. If V1 is CHARACTER or pictured-character, and if the output file is a PRINT file, PL/I places the characters in V2 in the output stream.
3. If V1 is CHARACTER or pictured-character, but the output file is not a PRINT file, PL/I places the following characters in the output stream: an apostrophe, all the characters in V2 but with each apostrophe in V2 replaced by two apostrophes in the output stream, and another apostrophe.
4. In all other cases, PL/I places the characters in V2 in the output stream.

The P Output Data Format Item

Purpose: Provides picture specifications for output.

Syntax: P'picture', where picture is any picture specification.

Rules: Let VI be the value of the scalar data item that is being printed with the P format item.

1. If the picture specification is pictured-numeric, PL/I converts VI to the numeric data type associated with the picture specification. PL/I then edits this numeric value into the character format associated with the picture specification, as described in Chapter 5. PL/I then inserts these characters in the output stream.
2. If the picture specification is pictured-character, PL/I converts VI to the CHARACTER data type, with a length equal to the length of the CHARACTER string value determined by the picture specification. If the resulting CHARACTER value is not valid for the picture specification, PL/I raises the CONVERSION condition. Otherwise, PL/I inserts these characters in the output stream.

Examples: Table 11-9 illustrates the P format item.

Table 11-9
The P Output Format Item

Line #	Data Value	Format Item	Output Chars
1	25	P'S9999'	+0025
2	25	P'-999V.ES999'	b250.E-001
3	25	P'X999'	bbb2
4	-25	P'X999'	(CONV error)

Lines 1 and 2 illustrate the P format item with a picture specification that is pictured-numeric. This is a very useful technique to get special kinds of output formats, such as the printing of leading plus signs or leading zeros.

Lines 3 and 4 illustrate the pictured-character case. This case is not very common in practice, since the A format item is often more useful. In line 3, PL/I converts the value 25 to the CHARACTER value 'bbb25', which it truncates to 'bbb2', because the length of the string

associated with the picture specification for P'X999' is 4. PL/I prints these four characters. On the other hand, in line 4, PL/I converts the value -25 to the CHARACTER value 'bb-25', which it then truncates to 'bb-2'. Since this string is not valid for the picture specification, PL/I raises the CONVERSION condition.

For more information on picture specifications and their associated numeric and string data types, see Chapter 5, DATA TYPES AND DATA ATTRIBUTES.

The PAGE Output Option and Control Format Item

Purpose: Starts a new page.

Syntax: PAGE

PAGE is both an option of the PUT statement and a format item used with PUT EDIT. The PAGE option is legal only if the file has the PRINT attribute.

Rules: PL/I skips to the top of a new printer page on the output file and resets its internal line number count for the output page to 1. PL/I does not raise the ENDPAGE condition.

Examples:

```
PUT EDIT(X, Y) (F(5), PAGE, F(5));
```

PL/I prints the values of X and Y in the same format, F(5), but on separate pages of output. After that, the statement

```
PUT PAGE LIST(Z);
```

prints the value of Z on a third page.

The R Output Format Item

Purpose: Identifies the format item to be used.

Syntax: R(reference)

This is the remote format item described earlier in the section on Matching Data Items to Format Items. The reference is a format variable or format constant.

The SKIP Output Option and Control Format Item

Purpose: Starts a new line or record.

Syntax: SKIP or SKIP(n), where n, if specified, is any PL/I expression. SKIP is both an option of the PUT statement and a format item used with PUT EDIT. PL/I evaluates n and converts it to an integer. It is an error for n to be negative or zero. If n is not specified, let $n = 1$.

Rules: PL/I moves to the beginning of a new output record (print line) n times. For ASCII files, PL/I accomplishes this by placing n carriage return characters in the output stream.

The STRING Output Option

Purpose: Moves data to a string variable.

Syntax: STRING(reference), where the reference is a scalar variable, possibly a structure or array element, with the CHARACTER attribute, either VARYING or NONVARYING. The reference may be BASED. Also, the reference may be to one of the ONCHAR, ONSOURCE, or SUBSTR pseudovariables.

Rules: PUT with the STRING option is no longer an output statement in the narrow sense of transmitting data to an external device. Instead, PUT STRING is a purely computational statement that manipulates internal data. When you use the STRING option, the stream of characters that PL/I would normally transmit to a file is instead stored in the reference variable. If the reference variable is not large enough to handle all the characters that would normally be printed by the PUT statement to a file, PL/I raises the ERROR condition.

Examples:

```

DECLARE (C, D) CHARACTER(200) VARYING;
DECLARE (X INITIAL(5), Y INITIAL(10.3)) FLOAT DECIMAL(5);
PUT STRING(C) EDIT(X, Y) (F(2), F(7,2));
PUT STRING(D) DATA(X);

```

Each of these PUT statements specifies output to a string variable, C or D. The resulting values are

```

C = 'b5bb10.30'
D = 'X = b1.0300E+01;'

```

The TAB Output Control Format Item

Purpose: Puts output at the desired tab stop.

Syntax: TAB or TAB(n), where n if specified, is any PL/I expression. PL/I evaluates n and converts it to an integer. It is an error for n to be negative.

Rules: If n is not specified, let $n = 1$.

1. If $n = 0$, PL/I does nothing.
2. If there are at least n tab positions remaining on the current output line (as set either by default or explicitly by the TAB option of the OPEN statement), PL/I prints a sufficient number of blank characters to move n tab stop positions on the output line.
3. If there are not at least n tab positions remaining on the current line, PL/I performs a PUT SKIP to the output file, and if $n > 1$, PL/I moves to the first tab position on the line.

Example:

```

PUT EDIT(X, Y) (SKIP, TAB, F(5), TAB(2), F(2));

```

PL/I prints the value of X after the first tab position in the output file and skips two tab positions to print Y.

The X Output Control Format Item

Purpose: Provides blanks in output.

Syntax: X(n), where n is any PL/I expression. PL/I evaluates n and converts it to an integer. It is an error for n to be negative.

Rules: If n = 0, PL/I does nothing; otherwise, PL/I places n blank characters in the output stream.

DETAILED SPECIFICATIONS FOR THE GET STATEMENT

This section gives the detailed specifications for all the options of the GET statement and all of the format items of the GET EDIT statement. The options and format items are listed together in alphabetical order. Some keywords, such as SKIP, are both options and format items.

For each option and format item, the list gives the syntax and rules. The rules explain how many characters are taken from the input stream and how they are interpreted. Note that, in taking characters from the input stream, PL/I ignores record boundaries, unless the rules for the option or format item specify otherwise.

The A Input Data Format Item

Purpose: Accepts input as a CHARACTER string.

Syntax: A or A(w), where w, if specified, is any PL/I expression. PL/I evaluates w and converts it to an integer. It is an error for w to be negative.

Note

The form A, with w not specified, is permitted by Prime PL/I, but is forbidden by ANS PL/I rules and so may not be valid in other implementations of the PL/I language. However, this is the only way to use variable-length edited input. For variable-length unedited input, consider using the nonstandard READ statement, described earlier in READ and WRITE With STREAM I/O.

Rules:

1. If w is specified, PL/I takes w characters from the input stream and forms a CHARACTER NONVARYING string from these characters.
2. If w is not specified (this is the nonstandard case), PL/I takes all remaining characters in the current input record from the input stream, and forms a CHARACTER NONVARYING string of these characters.

In either case, PL/I converts the string to the data type of the target variable according to the rules in Chapter 6, and assigns the result to that target variable.

Examples: Table 11-10 illustrates the A format item. In line 1, four characters are taken from the input stream, and the resulting value is the CHARACTER(4) NONVARYING string 'ABCD'. In line 2, the format item is A(0), which means that no characters are taken from the input stream, and the resulting input value is the null string. In line 3, we assume that the remaining characters in the input record are '12.34'.

Table 11-10
The A Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	A(4)	ABCD	'ABCD'
2	A(0)	(none)	''
3	A	12.34	'12.34'

The B, B1, B2, B3, and B4 Input Data Format Items

Purpose: Accepts input as a BIT string of the specified radix.

Syntax: B(w), B1(w), B2(w), B3(w), or B4(w), where w is any PL/I expression. PL/I evaluates w and converts it to an integer. It is an error for w to be negative.

Rules: In the following, let n be the radix factor of the format item. That is, the format item is B n (w), where $n = 1, 2, 3, \text{ or } 4$, and where B is the same as B1.

PL/I Reference Guide

1. PL/I takes w characters from the input stream.
2. Using Table 11-11, PL/I converts each of these w characters into a group of n bits. If any character is not listed in the Input Character column of the table, or if the corresponding table entry is illegal, PL/I raises the CONVERSION condition.
3. Using the w groups of n bits each, PL/I forms a single BIT NONVARYING string of $n * w$ bits.
4. Using the conversion rules in Chapter 6, PL/I converts this BIT string to the data type of the target variable and assigns the result to the target variable.

Table 11-11
Characters Corresponding to the B(n) Format Item

Input Char	Formation			
	B or B1	B2	B3	B4
0	0	00	000	0000
1	1	01	001	0001
2	Invalid	10	010	0010
3	Invalid	11	011	0011
4	Invalid	Invalid	100	0100
5	Invalid	Invalid	101	0101
6	Invalid	Invalid	110	0110
7	Invalid	Invalid	111	0111
8	Invalid	Invalid	Invalid	1000
9	Invalid	Invalid	Invalid	1001
A	Invalid	Invalid	Invalid	1010
B	Invalid	Invalid	Invalid	1011
C	Invalid	Invalid	Invalid	1100
D	Invalid	Invalid	Invalid	1101
E	Invalid	Invalid	Invalid	1110
F	Invalid	Invalid	Invalid	1111

Example: Table 11-12 illustrates the various bit format items.

Table 11-12
The B Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	B(5)	01101	'01101'B
2	B(7)	01101bb	(CONV error)
3	B(0)	(none)	''B
4	B1(5)	01101	'01101'B
5	B2(3)	123	'011011'B
6	B2(4)	4123	(CONV error)
7	B2(0)	(none)	''B
8	B3(3)	351	'011101001'B
9	B3(2)	34	'011100'B
10	B4(2)	58	'01011000'B
11	B4(2)	DE	'11011110'B

Lines 1 through 3 illustrate B(w). In line 1, the format item B(5) causes PL/I to take the five characters 01101 from the input stream. The resulting input value is the BIT NONVARYING string, '01101'B. In line 2, PL/I takes seven characters from the input stream, as a result of the B(7) format item. Since these seven characters include the b character, which is illegal for the B format item, PL/I raises the CONVERSION condition. In line 3, there are no input characters for the B(0) format item, and the resulting input value is the null BIT string.

Line 4 illustrates the B1 format item, which is the same as B.

Lines 5 through 7 illustrate the B2(w) format item. In line 5, because of the B2(3) format item, PL/I takes three characters, 123, from the input stream. Using Table 11-11, PL/I converts these three characters into three groups of two bits each, 01, 10, and 11, respectively, and so the resulting BIT string value is '011011'B. In line 6, the first input character is 4, and so there is a CONVERSION error, since this character is not permitted for B2. In line 7, since w equals 0, PL/I takes no characters from the input stream, and the resulting input value is the null BIT string.

Lines 8 through 11 illustrate B3 and B4. The rules are similar to the rules for B2.

The C Input Data Format Item

Purpose: Accepts input as a complex number.

Syntax: C(form1) or C(form1, form2), where form1 and form2 are each data format items of one of the following types: F, E, or P. If P, then the picture specification must be pictured-numeric, rather than pictured-character. If form2 is not specified, let form2 equal form1.

Rules:

1. PL/I inputs a data value, vr, according to the format form1.
2. PL/I inputs a data value, vi, according to the format form2.
3. PL/I creates a single COMPLEX value, vc, by computing

$$vc = \text{COMPLEX}(vr, vi);$$

The data type of vc depends upon the data types of vr and vi according to the rules of the COMPLEX built-in function.

4. PL/I converts vc to the data type of the target variable, and assigns the results to the target variable.

Examples: Table 11-13 illustrates the C format item for input.

Table 11-13
The C Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	C(F(2))	b4b3	4+3I
2	C(F(2))	42b3	42+03I
3	C(F(1), F(3))	1.23	1.00+0.23I
4	C(E(4), E(4))	1E0b0.23	1.00E0+2.30E-01I

In line 1, PL/I uses F(2) to input each of the real and imaginary parts. The first two characters are b4, as a result of which vr equals 4 with a data type of REAL FIXED DECIMAL(1,0). The next two characters are b3, so that vi equals 3, with the same data type. Using the rules for the COMPLEX built-in function, the combined data type is COMPLEX FIXED DECIMAL(1,0), and the resulting value for vc is 4 + 3I.

In line 2, the imaginary part of the input is the same, but the first two characters are 42, and so vr equals 42, with a data type of REAL FIXED DECIMAL (2,0). The combined data type is COMPLEX FIXED DECIMAL(2,0), with a resulting value for vc of 42 + 03I.

Lines 3 and 4 are similar, except that the real and imaginary parts have different format items. In line 4, PL/I uses the format item E(4) to input the four characters 1E0b from the input stream, with the result that vr equals 1E0, with the data type REAL FLOAT DECIMAL(1). PL/I uses E(4) to input the characters 0.23, and get a resulting imaginary value vi of 0.23, in the data type REAL FIXED DECIMAL(3,2). The resulting combined data type vc is COMPLEX FLOAT DECIMAL(3), with the value of 1.00E0+2.30E-01I.

The COLUMN Input Control Format Item

Purpose: Accepts input from the desired column.

Syntax: COLUMN(n), where n is any PL/I expression. PL/I evaluates n and converts it to an integer. It is an error for n to be negative.

Abbreviation: COL for COLUMN

Rules: The intention is that, after execution of the COLUMN format item, the next stream character will come from the nth column of the input record. If the previous input operation has taken n or more characters from the current input record, PL/I performs a GET SKIP before moving to column n.

Therefore, if c characters have already been taken from the current record, then

1. If $c < n$, PL/I takes $(n - c - 1)$ characters from the current record in the input stream.
2. If $c \geq n$, PL/I performs a GET SKIP operation, then takes $(n - 1)$ characters from the input stream.

Examples:

```
GET EDIT(X, Y, Z) (COLUMN(1), F(3), COL(70), F(5));
```

PL/I inputs the value of X starting in column 1, and the value of Y starting in column 70. PL/I inputs the value of Z starting in column 1 of the next record.

The COPY Input Option

Purpose: Copies input to a file.

Syntax: COPY(file-reference) or COPY, where file-reference is any FILE constant or FILE variable or an expression with a data type of FILE. If the file-reference is not specified, PL/I uses COPY(SYSPRINT).

Rules: All characters that PL/I takes from the input stream in order to execute the GET statement are printed in the file specified by the COPY option. This includes characters that are skipped, such as characters skipped because of the X format item for GET EDIT.

Examples: The statement

```
GET COPY LIST(X,Y);
```

inputs values for X and Y from the SYSIN file, printing the input stream characters on the file SYSPRINT.

The statement

```
GET FILE(TAPEIN) COPY(TAPEOUT) EDIT(A) (X(5), F(2));
```

takes seven characters as input from TAPEIN, and transmits them to file TAPEOUT.

The DATA Input Option

Purpose: Accepts input values with data names.

Syntax: DATA(list) or DATA

The method for determining the individual data items for the list has been described in the section on The GET DATA Statement earlier in this chapter. The list may contain only level-1 identifiers that are not BASED.

If you specify no list, all non-BASED variables accessible to the GET statement, including those declared in containing blocks, are in the implied input list.

Rules: The input stream must look like a list of assignments of the form

reference = constant

where the reference is a reference to a scalar element in the list, and the constant is any PL/I computational constant. The reference may be one of the items appearing in the list, or it may be a scalar element of such an item, if the item is an aggregate. If the reference is to an array element, all subscripts must be decimal integer constants.

The individual assignments must be separated in the input stream by a comma, one or more blank characters, or both. The last assignment in the input stream should terminate with a semicolon.

Examples: See the section, The GET DATA Statement, earlier in this chapter.

The E Input Data Format Item

Purpose: Accepts input as scientific notation.

Syntax: E(w), or E(w, d), or E(w, d, s), where each of w, d, and s, if specified, is any PL/I expression. PL/I evaluates each argument specified and converts it to an integer. If s is specified, PL/I ignores it. If d is not specified, let d equal 0. It is an error if w is negative. PL/I ignores s, if specified.

Rules: PL/I takes w characters from the input stream, to form a CHARACTER string of length w. This CHARACTER string must be in one of the following formats:

- A null string (if w = 0), or all blank characters. (This case is interpreted as a zero value.)
- A decimal constant in the format

[b . . . b] [+] 9 . . . 9 [.] 9 . . . 9 [b . . . b]

where 9 stands for a decimal digit. The leading blank characters, trailing blank characters, sign, and decimal point are all optional.

- A scientific notation constant in the format

[b . . . b][+ 9 . . . 9 [.] 9 . . . 9 E [+] 9 . . . 9 [b . . . b]

where 9 stands for a decimal digit. The leading and trailing blank characters, both signs, and the decimal point are all optional.

- A scientific notation constant in the format

[b . . . b] [+] 9 . . . [9 . 9] . . . 9 + 9 . . . 9 [b . . . b]

where 9 stands for a decimal digit. The leading and trailing blank characters, the first sign, and the decimal point are all optional. Since the E separating the mantissa from the characteristic is omitted in this format, the sign of the characteristic is required.

PL/I obtains a numeric value and data type according to one of the following rules:

1. If the CHARACTER string is the null string or is all blank characters (case 1 above), PL/I changes it to '0'.
2. If the CHARACTER string is a scientific notation constant with no E (last format above), PL/I changes the CHARACTER string by inserting an E between the characteristic and the mantissa portions of the constant.
3. PL/I converts the CHARACTER string to the numeric data type appropriate to the constant. In case 1 above, the data type is REAL FIXED DECIMAL(1,0). In case 2, the data type is REAL FIXED DECIMAL(p, q), where p is the number of decimal digits in the constant, and q is the number of digits following the decimal point. In cases 3 and 4, the data type is REAL FLOAT DECIMAL(p), where p is the number of decimal digits in the mantissa portion of the constant. Note that the values of p and q do not depend on the value of w, as specified in the format item.
4. If the CHARACTER string contains no decimal point, PL/I multiplies the value obtained in the previous step by 10^{-d} , and, if the data type is REAL FIXED DECIMAL(p, q), PL/I increases the value of q by d.

PL/I converts the resulting numeric value to the data type of the target variable, using the rules in Chapter 6, and assigns that value to the target variable.

Examples: Table 11-14 illustrates the various forms of the E format item.

Table 11-14
The E Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	E(8)	bbbbbb23	+23
2	E(8)	b23.4bbb	+23.4
3	E(8)	+2348E-1	+2.348E+02
4	E(8)	+2348-01	+2.348E+02
5	E(8)	bbbbbbbbb	+0
6	E(4)	-123	-123
7	E(4,1)	-123	-12.3
8	E(4,3)	-123	-.123
9	E(4,-1)	-123	-123F1
10	E(5,3)	-123.	-123
11	E(6)	234.E0	+2.34E+02
12	E(6,2)	234.E0	+2.34E+02
13	E(5,2)	234E0	+2.34E+00
14	E(5,2,25)	234E0	+2.34E+00

Lines 1 through 5 illustrate various types of input with the E(8) format item. The data type of the resulting numeric value depends on the characters taken as input from the input string. In line 1, PL/I treats bbbbb23 as the REAL FIXED DECIMAL(2,0) value +23, while in line 2, PL/I converts b23.4bbb to the REAL FIXED DECIMAL(3,1) value +23.4. Line 3 illustrates a FLOAT value, and line 4 illustrates FLOAT with no E separating the mantissa from the characteristic. In line 5, the input is all blanks, and the resulting numeric value is the REAL FIXED DECIMAL(1,0) value, 0.

Lines 6 through 10 illustrate what happens when the format item specifies a value for d and the input stream value has a FIXED data type. The effect is to move the decimal point in the value to the left d places, and to increase the scale factor in the data type by the amount d, provided that the input characters have no decimal point. For example, in line 7, the input stream contains the characters -123, which PL/I converts to the value -123 with the data type REAL FIXED DECIMAL(3,0). However, since the format item has a value of 1 for d, the value is changed to -12.3 (PL/I moves the decimal point to the left one place), and the data type becomes REAL FIXED DECIMAL(3,1). In line 10, the input stream contains a decimal point, and so the value 3 for d has no effect.

Lines 11 through 13 illustrate the effect of specifying d with a FLOAT input stream. If there is no decimal point in the input stream, PL/I moves the decimal point to the left d places in the input value. This has the effect of decreasing the value of the characteristic by d.

As line 14 illustrates, when a third argument to the E format item is specified, PL/I ignores it.

The EDIT Input Option

Purpose: Accepts edited input.

Syntax: EDIT(reference-list)(format-list), or
EDIT(reference-list)(format-list)(reference-list)(format-list)...

The rules for determining the individual data items for the reference-list have been described in the section on The GET EDIT Statement. The reference-list may include aggregates and implied DO loops. The individual data items may not be arbitrary expressions, but must be variables or elements of aggregates.

The EDIT option may specify two or more reference-list and format-list pairs. PL/I matches the data items in the first reference-list to the format items in the first format-list, the data items in the second reference-list to the format items in the second format-list, and so forth.

The F Input Data Format Item

Purpose: Accepts edited numeric input.

Syntax: F(w), or F(w, d), or F(w, d, s), where each w, d, and s is any PL/I expression. PL/I evaluates each operand specified and converts it to an integer. If d is not specified, let d equal 0. If s is not specified, let s equal 0.

Rules: PL/I takes w characters from the input stream and forms a CHARACTER string of length w. The string must be in one of the following formats:

- The null string (if w = 0) or all blank characters. (PL/I interprets such a string as a zero value.)
- A decimal constant in the format

[b . . . b] [+] 9 . . . 9 [.] 9 . . . 9 [b . . . b]

where 9 stands for a decimal digit. The leading and trailing blank characters, the sign and the decimal point are all optional.

PL/I obtains a numeric value and data type from the CHARACTER string as follows:

1. If the CHARACTER string is the null string, or is all blank characters (case 1 above), PL/I changes the CHARACTER string to '0'.
2. PL/I converts the CHARACTER string to the numeric data type appropriate to the constant. The data type is REAL FIXED DECIMAL(p , q), where p is the number of decimal digits in the CHARACTER string, and q is the number of those digits that follow the decimal point, if any.
3. If the CHARACTER string contains no decimal point, PL/I multiplies this numeric value by 10^{-d} , and increases the value of q by d .
4. In all cases, PL/I multiplies the value by 10^s , and decreases the value of q by s .

PL/I converts the resulting numeric value to the data type of the target variable using the rules in Chapter 6, and assigns that value to the target variable.

Examples: Table 11-15 illustrates the F format item.

Table 11-15
The F Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	F(4)	b23b	+23
2	F(4)	23.4	+23.4
3	F(4)	-14b	-14
4	F(4)	bbbb	+0
5	F(4)	-123	-123
6	F(4,1)	-123	-12.3
7	F(4,3)	-123	-.123
8	F(4,-1)	-123	-123F1
9	F(5,3)	-123.	-123
10	F(4,3,1)	-123	-1.23
11	F(5,3,1)	-1.23	-12.3

PL/I Reference Guide

Lines 1 through 5 illustrate the F(4) format item, with various values in the input stream. In line 1, the input stream contains b23b, which PL/I converts to a numeric value of +23, with a data type of REAL FIXED DECIMAL(2,0). In line 4, where the input stream is all blank characters, the resulting input value is 0, with a data type of REAL FIXED DECIMAL(1,0).

Lines 5 through 9 illustrate what happens when the F format item has a second argument, d. The value of d has no effect if the input stream contains a decimal point; but if the input stream contains no decimal point, a nonzero value of d can change both the value and the data type of the input value. If the value of d is nonzero, PL/I moves the decimal point in the value to the left d places (to the right if d is negative), and increases the scale factor in the data type by the value of d.

In line 6, PL/I converts the input stream characters -123 to the value -123, with a data type of REAL FIXED DECIMAL(3,0). But since the input stream characters contain no decimal points and since d = 1, PL/I moves the decimal point in the value to the left one place, and increases the scale factor by 1. The resulting input value is -12.3, with a data type of REAL FIXED DECIMAL(3,1). The other examples are similar. In line 9, the input stream characters contain a decimal point, so the value of d is ignored.

Lines 10 and 11 illustrate what happens when you specify a third argument, s, to the F format item. PL/I moves the decimal point to the right s places and increases the scale factor of the data type by s. Therefore, the effect of s is the same as the effect for d with the sign reversed, except that the effect of s is felt even if the input stream contains a decimal point.

In line 10, the input stream is -123, which PL/I converts to the value -123 with a data type of REAL FIXED DECIMAL(3,0). Since there is no decimal point in the input stream characters, the effect of d = 3 is to change the value to -.123, with a data type of REAL FIXED DECIMAL(3,3). Then, since s = 1, the numeric result is changed again to -1.23, and the data type is changed to REAL FIXED DECIMAL(3,2).

In line 11, the input stream contains a decimal point, so that the value of d is ignored. Since s = 1, the REAL FIXED DECIMAL(3,2) value -1.23 is changed to a REAL FIXED DECIMAL(3,1) with the value -12.3.

The FILE Input Option

Purpose: Specifies the file from which input is taken.

Syntax: FILE(reference), where the reference is an expression whose data type is FILE. Usually the reference is to a FILE constant, as explained in the section on PUT and GET to files and devices, but it may also be a FILE variable or a FILE value returned by a user-defined

function. FILE variables and user-defined functions are explained in Chapter 12, RECORD INPUT/OUTPUT.

The LIST Input Option

Purpose: Accepts character input with separators.

Syntax: LIST(data-list)

The data-list can contain references to variables, including aggregates, aggregate elements, and implied DO groups. The individual data items are established as described in the section ESTABLISHING DATA ITEMS. Each scalar reference must have a computational data type.

Rules: For each input data item, PL/I must take a value from the input stream and assign it to the data item. This requires the following steps:

1. PL/I skips over any separators between data items in the input stream.
2. PL/I takes a string of characters from the input stream and interprets that string of characters as an input value. In most cases, the input value is a CHARACTER string; however, in some cases, the input value is interpreted as a BIT string.
3. PL/I converts the CHARACTER BIT string to the data type of the input data item and assigns the results to the input data item.

The rules for these steps are described in the following paragraphs.

The separators mentioned in Step 1 can be commas or blanks. GET LIST data values in an input stream must be separated either by one or more blanks, by a single comma, or by both a comma and one or more blanks. Thus, for example, if you had the values 2 and 3 in your input stream, you could separate them in any of the following ways:

2b3	2,bb3
2,3	2bbbb,3
2bbbb3	2bb,bbbb3

When PL/I completes the input operation for one GET LIST data value (2 in the above example), it stops taking input characters after it has taken the blank or comma following that data value. PL/I does not skip over the full separator (the remaining blanks or comma), until the next GET LIST input request (for 3 in the above example). Therefore, when PL/I begins input for a new GET LIST value, it must remember whether the terminator for the previous GET LIST value was a blank or a comma.

In order for PL/I to skip over the separator for the new value it must skip over blank characters, searching for the first nonblank character; but if the preceding GET LIST value was terminated by a blank, rather than by a comma, PL/I can skip over a single comma in the separator as well.

Note that if, after PL/I has skipped over one comma, it finds that the first nonblank is a second comma, then PL/I considers the input value to be missing from the input stream. In this case, PL/I leaves the value of the target data item unchanged. Consider, for example, the following statements:

```
X, Y = 2;  
GET LIST(X, Y);
```

Suppose that when PL/I executes the GET statement, it finds the following characters in the input stream:

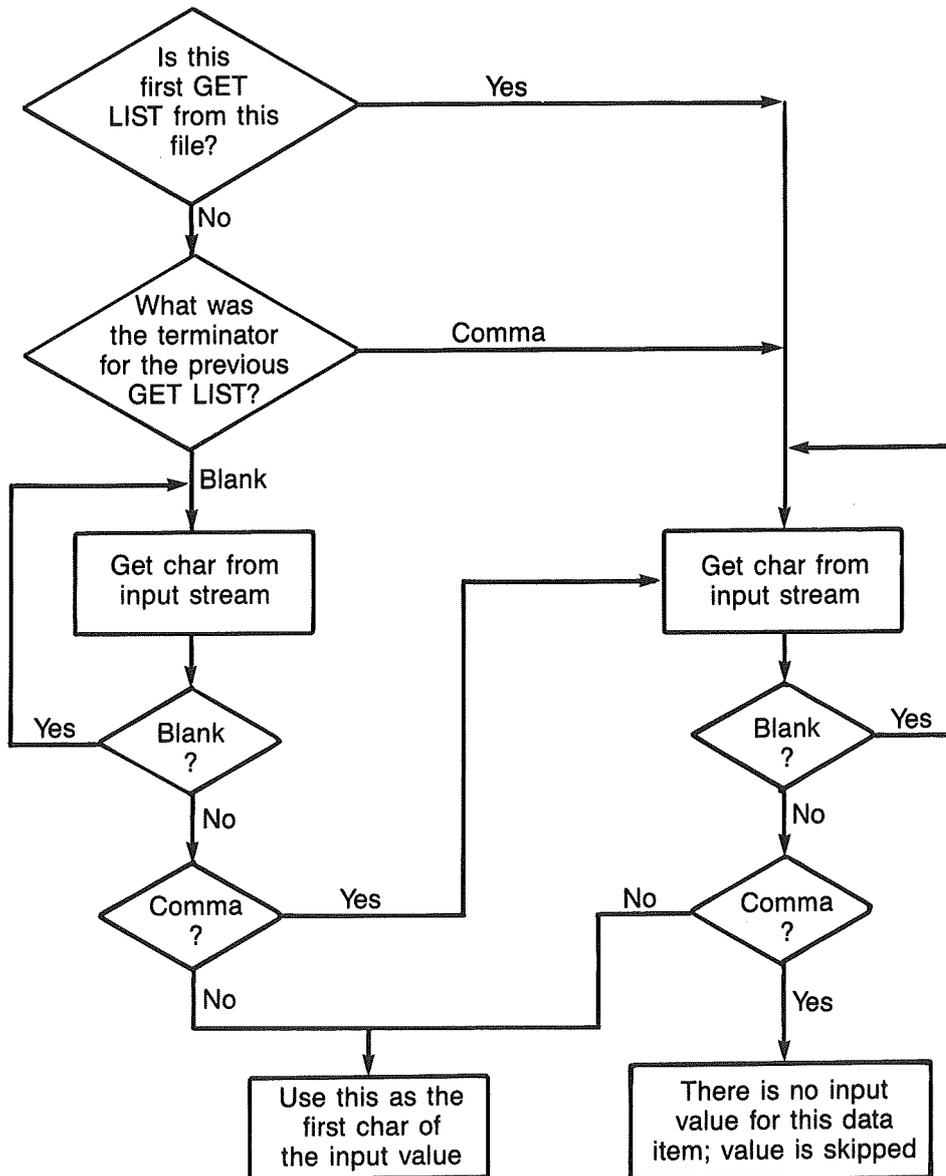
```
3,,
```

Then PL/I considers the first input data value to be 3, and the second input data value to be missing. As a result, PL/I sets X to 3, and leaves Y with its old value of 2.

Figure 11-1 summarizes the algorithm that PL/I uses to skip over the separator between GET LIST data items. At the point when control enters the logic specified by this figure, PL/I is ready to begin skipping over the separator for the next GET LIST input value. When this logic is finished, PL/I either has decided that the input value is skipped, or has found the first character of the input value.

Now let us turn our attention to how PL/I identifies the input data value from the input stream. There are two major cases:

- If the first character of the input data value after the separator is not an apostrophe, PL/I takes characters from the input stream, stopping at the first blank or comma, and forms a CHARACTER string of these characters. Lines 1 through 4 of Table 11-16 illustrate this case.



Handling Separators with GET LIST
Figure 11-1

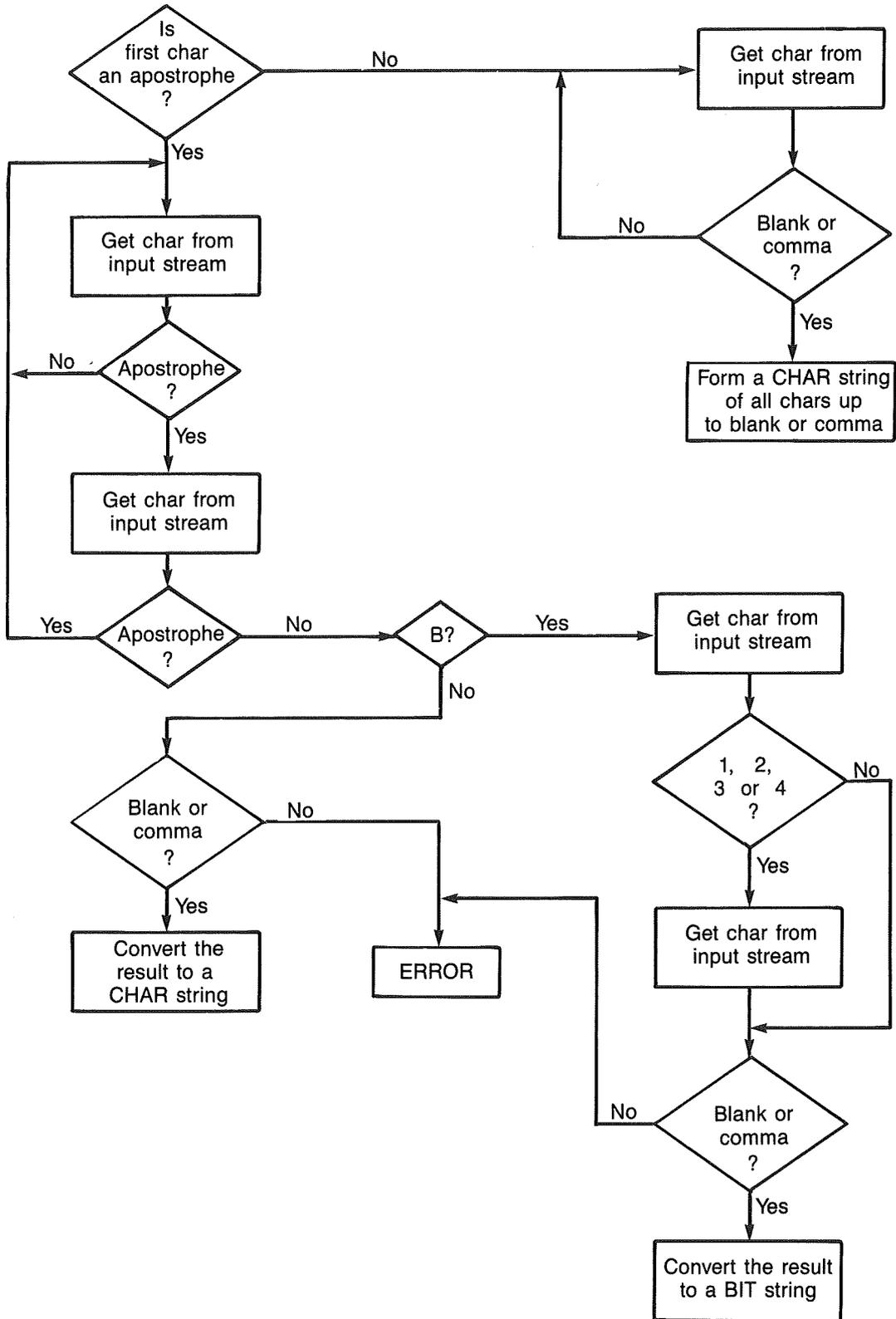
Table 11-16
The LIST Input Option

Line #	Input Stream	Input Value
1	764b	'764'
2	48E+23,	'48E+23'
3	7E2+16E3I,	'7E2+16E3I'
4	ABCDEFb	'ABCDEF'
5	'ABCD,bF'b	'ABCD,bF'
6	'DON''TbGO',	'DON''TbGO',
7	'1011'B,	'1011'B
8	'34'B3	'011100'B
9	'A'B4,	'1010'B
10	'ABC'X,	(error)
11	'ABC'B3,	(error)
12	'011'BXY,	(error)

- If the first character after the separator is an apostrophe, PL/I takes characters until it finds the matching apostrophe. In searching for the matching apostrophe, PL/I must correctly handle the situation where two apostrophes in a row appear in the input stream with the usual meaning. The matching apostrophe may be followed by B or by B_n, where the _n is a radix factor; this is precisely the situation where the input value would be considered a BIT value rather than a CHARACTER value. In any case, the CHARACTER or BIT string constant must end with a blank or comma. Lines 5 through 9 of Table 11-16 illustrate these cases, while lines 10 through 12 illustrate error situations.

Figure 11-2 illustrates these rules. The logic in this figure follows immediately after the logic in the preceding figure, assuming that the search for the end of the separator has ended with the finding of the first character of the new data value. When the logic in this new figure has been completed, PL/I has identified the entire input data value and determined whether it is a CHARACTER value or a BIT value.

In any case, PL/I ends up with a CHARACTER or BIT string value. PL/I converts this value to the data type of the target data item, following the rules in Chapter 6, and assigns the result to the target data item.



Identifying the Input Value With GET LIST
Figure 11-2

The P Input Data Format Item

Purpose: Accepts input with picture specifications.

Syntax: P'specification', where specification is a picture specification, either pictured-numeric or pictured-character.

Rules: PL/I takes from the input stream a number of characters equal to the length of the CHARACTER string value corresponding to the picture specification. This string of input characters must be valid for the picture specification; otherwise, PL/I signals the CONVERSION condition.

Further action depends on whether the picture specification is pictured-string or pictured-numeric.

1. If the picture specification is pictured-string, PL/I forms the input characters into a CHARACTER string, which it uses as the input value.
2. If the picture specification is pictured-numeric, an extra conversion takes place. PL/I converts the string of characters to the numeric data type corresponding to the picture specification, and that numeric value is used as the input value.

In either case, PL/I converts the resulting string or numeric value to the data type of the target variable, and assigns the result to the target variable.

Example: Table 11-17 illustrates the P format item. Lines 1 and 2 illustrate a specification for pictured-string, while lines 3 and 4 illustrate pictured-numeric.

Table 11-17
The P Input Format Item

Line #	Format Item	Input Stream	Resulting Input Value
1	P'AX99'	M-23	'M-23'
2	P'AX99'	1234	(error)
3	P'99V.9'	23.4	23.4
4	P'99V.9'	23,4	(error)

Lines 2 and 4 are error situations because the input stream characters are not valid for the picture specification specified with the P format item. See Chapter 5 for further information on picture specifications.

The R Input Format Item

Purpose: Specifies the format item to be used with input.

Syntax: R(reference).

This is the remote format item, and it has already been discussed in the section on MATCHING DATA ITEMS TO FORMAT ITEMS.

The SKIP Input Option and Control Format Item

Purpose: Moves to a new line or record of input.

Syntax: SKIP or SKIP(n), where n, if specified, is any PL/I expression. SKIP is both an option of the GET statement and a format item used with the GET EDIT statement. PL/I evaluates n and converts it to an integer. It is an error for n to be zero or negative. If n is not specified, let n = 1.

Rules: PL/I moves to the beginning of a new input record n times. For ASCII files, PL/I accomplishes this by taking characters from the input stream, and stopping after n carriage return characters have been accepted.

The STRING Input Option

Purpose: Moves data from a string variable.

Syntax: STRING(expression), where the expression is a scalar expression. PL/I converts the expression to the CHARACTER data type.

Rules: GET with the STRING option is no longer an input statement in the narrow sense of transmitting data from an external device. Instead, GET STRING is a purely computational statement which manipulates internal data.

When you use the STRING option, the stream of characters that would normally be transmitted from a file or device is taken instead from the value of the expression.

If the CHARACTER string value of the expression is not long enough to satisfy the input stream requirements for the GET statement, PL/I signals the ERROR condition. In particular, for GET LIST with the STRING option, the CHARACTER string expression must contain the blank or comma that terminates the input data value.

Examples:

```
DECLARE C CHARACTER(4) INITIAL('7642');  
GET STRING(C) EDIT(X, Y) (F(2));  
GET STRING(C||',') LIST(Z);
```

In each of the GET statements, the input stream is taken from the specified CHARACTER string expression, rather than from an input device like your terminal. The result is that X = 76, Y = 42, and Z = 7642. Note that in the last GET statement, the string expression is '7642,', including a comma concatenated to terminate the GET LIST value.

The X Input Control Format Item

Purpose: Skips input characters.

Syntax: X(n), where n is any PL/I expression. PL/I evaluates n and converts it to an integer. It is an error for n to be negative.

Rules: PL/I takes n characters from the input stream and ignores them.

12

RECORD

Input/Output

Standard STREAM input/output, discussed in Chapter 11, uses the GET and PUT statements to preserve machine and device independence by treating a file as a stream of characters. Chapter 11 also discusses use of nonstandard READ and WRITE statements with STREAM input/output.

RECORD input/output is a more general approach to I/O in that it permits you to read and write files and devices with any record format or organization. However, since RECORD I/O access methods reflect very closely the access methods supported by the Prime operating system, your program may not run on other implementations of PL/I. It is even possible that your program will work differently on different devices on the same computer.

CONCEPTS OF RECORD INPUT/OUTPUT

The easiest way to understand RECORD input/output is to contrast it with STREAM input/output. Standard STREAM input/output treats a file as a long stream of characters. The GET and PUT statements transmit characters to or from this stream. To accomplish this, PL/I must automatically convert internal data formats to and from the character representation that is stored in the STREAM file.

RECORD input/output considers a file to be a collection of records. Each READ or WRITE statement transmits precisely one record of the file to or from the I/O device. Furthermore, PL/I transmits that record directly from or to a block of program storage and performs no conversion during that transmission. Therefore, the records of the

file may contain character data, or they may contain data in any representation or format supported by the machine on which your PL/I program is running.

The precise meaning of a record depends upon the device to or from which data is being transmitted. Examples of records are as follows:

- If the file is a deck of punched cards, each card is considered to be a single record. In this case, therefore, each record contains precisely 80 characters.
- If the file is output to a line printer, each line printed is considered to be a record. Therefore, each record of a line printer file can contain up to 132 characters.
- If the file is stored on a disk or tape device, the records may be CHARACTER of varying or fixed length, or may be a fixed size, but containing data of arbitrary data types.

Sequential Versus Direct Access

RECORD input/output provides you with a direct access capability that is entirely unavailable with STREAM I/O. The direct access capability allows you to jump around in a file, reading or writing records in random order.

To understand this capability, imagine that you have a numbered list of names and addresses written on a sheet of paper. You cannot insert a new name and address between two existing names, since that would require renumbering the entire list after that point, which you do not wish to do. However, you may add a new name and address to the end of this list. (The resulting list, however, may not be in alphabetical order.)

The term direct access, when applied to such a list, means only that you can jump around the list, reading the entries in any order. For example, if you wish to read the fifteenth name in the list, there is no need for you to read the first 14 names first; instead, your eye can skim down to the fifteenth line, and you can read the information on that line. Furthermore, you may rewrite any line in the list, by erasing the information on that line and writing in a new name and address.

Now, however, suppose that you had such a file stored on magnetic tape. When a file is stored on magnetic tape, it may be accessed only by means of sequential operations. This means that your program could access such a file in any of the following ways:

- It could read the fifteenth record on the tape only by first reading the 14 records that precede it.

- It could add a new record to the end of the file by reading all the records in the file, and then writing a new record at the end of the file.
- It could change the fifteenth record of the file by positioning the tape just after the fourteenth record, and then rewriting the following record.

The important characteristic of this tape file is that you cannot go from one point to another in the file without reading all records in between.

When you put this file onto a direct access device, such as a disk, you have much more flexibility. PRIME provides two basic file organizations:

- SAM files (Sequential Access Method)
- DAM files (Direct Access Method)

Appendix I of the Subroutines Reference Guide contains more information about SAM and DAM files. Your program may access SAM files only sequentially, and they are treated the same as files on tape. Your program may handle a DAM file either by using the sequential techniques discussed above or by using direct access techniques.

Direct access techniques permit you to move from one point in the file to any other point directly, without having to read all the records in between. Therefore, the three operations described above for sequential access become the following with direct access:

- Your program can read the fifteenth record of the file directly, without having to access any of the records that precede it.
- Your program can add records to the end of the file without having to read the entire file first.
- Your program can change the fifteenth record of the file by rewriting it directly. It is not necessary to position the file by reading preceding records.

Therefore, use SAM files when you plan to access the records of the file only in the order in which they were stored into the file. If you plan to jump around in the file, use DAM files.

MIDASPLUS Files

PRIME provides a third file organization that gives you even more flexibility and power than SAM or DAM files. To understand this capability, let us go back to the list of names and addresses we discussed, but now suppose that you keep the information on 3 x 5 index cards rather than in a numbered list. If you have only one name and

address per index card, you can keep the cards in alphabetical order. The difference is that when you have to add a new name to your list, you can insert the new card in its proper position among the old cards; there is no need to put each new name and address at the end of the list. MIDASPLUS gives you the capability of inserting records anywhere in the file.

The major differences between DAM files and MIDASPLUS files are the following:

- You can access an arbitrary record in either file organization, but you specify the record in different ways. For DAM files, choose the record you want by specifying the number of the record according to its position in the file. For MIDASPLUS files, specify an arbitrary CHARACTER string as a key to the record. In the example of the list of names and addresses given above, you would choose your desired record by giving the person's name as a string of characters.
- You can add records to a file in either file organization, but for DAM files, you may add records only at the end. For MIDASPLUS files, you may insert records at any point within the file.
- For MIDASPLUS files, you may delete any record in the file. You may not delete records in a DAM file.

The Key of a Record

For direct access files, the DAM and MIDASPLUS organizations, each record of the file has a key. Use this key to specify which record of the file you wish to access.

For DAM files, the key of a record is an integer equal to the position of the record in the file. The first record of the file has a key of 1, the second has a key of 2, and so forth. The position of a record in the file is called the relative record number of the record within the file.

For MIDASPLUS files, the key for each record is an arbitrary string of up to 32 characters. The nature of this string of characters depends upon the particular application. For example, one application may use a person's name as the key for each record. A different application that keeps track of information about each town and city in the United States might use as a key the string of characters consisting of both the name of the city or town and the name of the state.

The PL/I interface to MIDASPLUS has no way of handling situations where two different records have the same alphabetic key. Therefore, if your application uses a person's name as a key, your application cannot handle two people with the same name. For this reason, many applications that keep records of different people use a unique

identification number (such as a social security number) as the key to each record, rather than the person's name.

SEQUENTIAL RECORD INPUT/OUTPUT

Let us begin with some examples of sequential mode RECORD I/O.

The Basic Statements

Figure 12-1 is a program that copies card images from a tape to a direct access file called 'Data'. In this program, the identifiers IN and OUT are FILE identifiers representing the input and output files, respectively. Each of these identifiers is given the following attributes:

- FILE is the data type attribute.
- RECORD specifies that your program will use RECORD rather than STREAM access.
- SEQUENTIAL specifies that SEQUENTIAL rather than DIRECT access is to be used.

```
COPY: PROC OPTIONS(MAIN);
      DCL (IN, OUT) FILE RECORD SEQUENTIAL;
      DECLARE CARD CHARACTER(80);
      OPEN FILE(IN) INPUT TITLE('MT0 -DEVICE -RECL 80');
      OPEN FILE(OUT) OUTPUT TITLE('DATA -DAM 80 -APPEND');
      ON ENDFILE(IN) FLAG = 1;
      FLAG = 0;
      READ FILE(IN) INTO(CARD);
      DO WHILE(FLAG = 0);
        WRITE FILE(OUT) FROM(CARD);
        READ FILE(IN) INTO(CARD);
      END;
      CLOSE FILE(IN), FILE(OUT);
      END COPY;
```

A Record I/O Program
Figure 12-1

These two file identifiers are used in all of the input/output statements. These statements are as follows:

1. The first I/O statement,

```
OPEN FILE(IN) INPUT TITLE('MT0 -DEVICE -RECL 80');
```

opens the file IN as an INPUT file and associates it with the device MT0 (a magnetic tape drive), with a record size of 40 words (80 characters) per record.

2. The statement

```
OPEN FILE(OUT) OUTPUT TITLE('DATA -DAM 80 -APPEND');
```

opens file OUT as an output file, associating it with a direct access file that has 80 characters, and positioning to append information to the end of the file.

3. The ON statement

```
ON ENDFILE(IN) FLAG = 1;
```

specifies what action is to be taken when end of file occurs on the input file, with file identifier IN. This statement says the following: If a READ statement fails because of end of file, then set FLAG equal to 1, and continue execution with the statement following the READ statement. The ON statement is described in detail in Chapter 13 on condition handling.

4. The statement

```
READ FILE(IN) INTO(CARD);
```

specifies that PL/I is to read one record from the file IN, and store the data in the variable called CARD. Since the OPEN statement described above has opened the file IN and connected it with a magnetic tape device, each READ statement reads one card image and stores the 80 characters in the CHARACTER(80) variable CARD.

5. The statement

```
WRITE FILE(OUT) FROM(CARD);
```

takes the data in the variable CARD and appends it to the end of the file DATA to which OUT has been attached by the second OPEN statement described above.

6. The final I/O statement

```
CLOSE FILE(IN), FILE(OUT);
```

is executed after end of file has been reached on file IN. This statement disconnects the file identifiers IN and OUT from the respective devices with which the OPEN statements have associated them.

In this program example, the DO loop works by reading each card image from a tape and writing it into the file DATA. The DO loop continues executing as long as the variable FLAG is equal to 0. FLAG is set to 1 by the on-unit specified in the ON ENDFILE statement when a READ statement fails because of end of file.

The IGNORE Clause

You may skip records in a SEQUENTIAL input or update file by using the IGNORE option of READ:

```
READ FILE(name) IGNORE(expr);
```

PL/I evaluates the expression, truncates it to an integer, and skips over that number of records. Thus, the statement

```
READ FILE(IN) IGNORE(1);
```

skips over one record.

Card Image Into CHARACTER Structure

Suppose you are writing a program to read cards from the card reader and you wish to process the information on these cards. Suppose further that each of the cards is punched in the following format:

Columns 1-20	Name
Columns 21-50	Address
Columns 51-60	Unused
Columns 61-68	Telephone # (xxx-xxxx)
Columns 69-80	Unused

By reading the cards into an appropriate structure, you can split the characters in the input card into their individual fields automatically. For example, suppose you used the following DECLARE statement:

```
DECLARE 1 CARD,  
      2 NAME CHAR(20) ,  
      2 ADDRESS CHAR(30) ,  
      2 UNUSED_1 CHAR(10) ,  
      2 PHONE CHAR(8) ,  
      2 UNUSED_2 CHAR(12) ;
```

If the identifier IN is declared and opened as in the preceding example, the statement

```
READ FILE(IN) INTO(CARD) ;
```

inputs a card image into the structure CARD. By then referencing the individual members of the CARD structure, you can directly reference the corresponding individual fields of each punched card.

Non-character Records

In all the previous examples of READ and WRITE, the records of the file or device contained only characters. Actually, data of any data type can be stored in such a file. For example, suppose your program contains the following declaration:

```
DECLARE 1 S(5) ,  
      2 A BIN FIXED ,  
      2 X(5) FLOAT DEC(3) ;
```

The array of structures S that is declared in this DECLARE statement can be used in the FROM option of the WRITE statement or in the INTO option of the READ statement. PL/I simply transmits the block of storage occupied by S to or from the input/output device. It performs no data conversions whatsoever.

RECORD Input/Output to Disk Files

Your program may access any files on disk, with arbitrary record format. For example, here are some statements that create a disk file:

```

DECLARE 1 REC,
        2 NAME CHAR(20),
        2 AGE FIXED BIN(15),
        2 SALARY FIXED DEC(10,2);
DECLARE F FILE RECORD SEQUENTIAL;
...
OPEN FILE(F) OUTPUT TITLE('NAMELIST -SAM 28');
...
WRITE FILE(F) FROM(REC);

```

In this example, the OPEN statement associates the file identifier F with the disk file NAMELIST, to be used as a SAM file with a record size of 14 words (28 bytes). If the file does not exist, NAMELIST is created. Each WRITE statement, such as the one shown in the example, transmits the block of storage occupied by the structure REC to a record in the file NAMELIST on disk.

Appending to an OUTPUT File

Suppose you open a disk file for OUTPUT, and suppose there is already a file on disk with the same file name. Ordinarily, PL/I creates a new file with that name, deleting the old file. However, if you use the APPEND option in the CHARACTER string specified with the TITLE option, PL/I does not replace the old file with the new file; instead, your new output records are appended to the end of the existing file.

In the previous example, if the OPEN statement were

```

OPEN FILE(F) OUTPUT TITLE('NAMELIST -APPEND 28');

```

PL/I would append the new records written by the WRITE statement to the end of the existing file NAMELIST.

DIRECT ACCESS WITH DAM FILES

The preceding examples illustrate files that are read or written sequentially; that is, the records of the file are accessed in order. For this reason, a SAM file is used.

If you wish to create a file that you can access by jumping around among the records, use a DAM file.

Creating a DAM File Sequentially

You may create a DAM disk file sequentially, in the same way that you create a SAM file. The only difference, when you create the file, is that you should specify DAM rather than SAM in the CHARACTER string that you specify with the TITLE option of the OPEN statement.

The last example, which created a SAM file, can thus be changed to

```
DECLARE 1 REC,  
        2 NAME CHAR(20),  
        2 AGE FIXED BIN(15),  
        2 SALARY FIXED DEC(10,2);  
DECLARE F FILE RECORD SEQUENTIAL;  
...  
OPEN FILE(F) OUTPUT TITLE('NAMELIST -DAM 28');  
...  
WRITE FILE(F) FROM(REC);
```

The only difference between this and the preceding example is that DAM, rather than SAM, is specified with the TITLE option of the OPEN statement. As before, each WRITE statement adds a new record to the end of the file. However, PL/I organizes the file differently on disk, so that it is possible to read and update the file more efficiently with direct access techniques. These techniques are illustrated below.

Reading a DAM File With KEYED DIRECT INPUT

The following explains DIRECT access techniques using keys. Any record of a DIRECT file has a key, a string of characters that you may use to access that record. In the case of DAM files, this string of characters contains an unsigned decimal integer, possibly with leading or trailing blanks. The value of this decimal integer is the relative record number of the record within the file. For example, you can reference the first record of the file with the key 1, and the 234th record of the file with the key 234.

The file created in the example of the preceding section illustrates how you can access a file using keys. This file was created

sequentially. To access the same file using direct access techniques, use an interactive program segment like the following, which allows the user to type in a record number and which then types out the corresponding record of the file:

```

DECLARE l S,
        2 NAME CHAR(20),
        2 AGE FIXED BIN(15),
        2 SALARY FIXED DEC(10,2);
DECLARE RECNUM FIXED DEC(5);
DECLARE F FILE RECORD KEYED DIRECT;
...
OPEN FILE(F) INPUT TITLE('NAMELIST -DAM 28');
PUT SKIP LIST('TYPE FIRST RECORD # DESIRED:');
GET LIST(RECNUM);
  DO WHILE(RECNUM > 0);
    READ FILE(F) INTO(S) KEY(RECNUM);
    PUT SKIP EDIT(S) (A, F(4), F(14,2));
    PUT SKIP LIST('TYPE NEXT REC # DESIRED');
    PUT LIST('(TYPE 0 TO TERMINATE):');
    GET LIST(RECNUM);
  END;

```

Compare this example with previous examples. Notice that the DECLARE statement for the identifier F specifies the attributes KEYED DIRECT rather than the SEQUENTIAL attribute that we have seen before. This means that you will use keys to jump around the file in direct access mode.

The READ statement in this example uses a FIXED variable RECNUM to specify the relative record number of the record in the file to be read. The statement

```
READ FILE(F) INTO(S) KEY(RECNUM);
```

executes by performing the following steps:

1. PL/I converts RECNUM to a CHARACTER string according to the rules in Chapter 6. For example, if RECNUM = 1623, PL/I converts RECNUM to the CHARACTER string 'bbbbbl623', where b stands for a blank character.
2. PL/I uses this CHARACTER string as a key to find the record with relative record number 1623 in the file.
3. PL/I reads that record from disk and transmits the data to the structure S, performing no conversion.

The new option in the READ statement is

```
KEY(RECNUM)
```

PL/I uses the variable RECNUM to determine which record you wish to read. Although PL/I converts the value of RECNUM from FIXED to CHARACTER, you may informally think of the key of a DAM file as the FIXED numeric value of RECNUM.

Updating a DAM File With KEYED DIRECT UPDATE

In all the examples so far, our OPEN statements have specified either the INPUT option to read from the file or the OUTPUT option to create the file or add records to the end of the file.

You may also specify UPDATE, instead of INPUT or OUTPUT. If you do that, subsequent statements of your program will be able to perform any of the following operations:

- Use a READ statement to input an arbitrary record of the file.
- Use a WRITE statement to add a new record to the end of the file.
- Use a REWRITE statement to change an existing record in the file.

Consider some specific examples. Suppose your PL/I program contains the following statements:

```
DECLARE F FILE RECORD KEYED DIRECT;  
OPEN FILE(F) UPDATE TITLE('NAMELIST -DAM 28');
```

This OPEN statement is the same as the OPEN statement in the preceding example except that it specifies the UPDATE option rather than either INPUT or OUTPUT.

Now assume that S and RECNUM are declared the same way in your program as they were in the last example. The statement

```
READ FILE(F) INTO(S) KEY(RECNUM);
```

causes PL/I to find the records specified by the FIXED variable RECNUM and to transmit the data from disk into the storage block occupied by S.

Since you have opened the file with the UPDATE attribute, you are permitted to do output to the file as well as input from the file. However, you must distinguish between two kinds of output operations: adding a new record to the file, and updating or changing an existing record. Use the WRITE statement to perform the first of these operations and the REWRITE statement to perform the second.

If RECNUM contains the number of the record that you wish to change, use the statement

```
REWRITE FILE(F) FROM(S) KEY(RECNUM);
```

to change the existing record. In this case, you use the block of storage occupied by the variable S to replace the data in the record with relative record number RECNUM. The old data in that record is lost.

If, however, you wish to add a new record to the end of the file, use the WRITE statement. An example is

```
WRITE FILE(F) FROM(S) KEYFROM(RECNUM);
```

When you use this statement, RECNUM should have a value larger than the number of records that you have already written to the file. If there are n records in the file before your program executes this statement, this WRITE statement adds $\text{RECNUM} - n$ new records to the end of the file. PL/I copies the storage block occupied by S into each of those new records.

In fact, such use of the WRITE statement shows an easy way to expand an existing file. Suppose you wish to expand this file from its current size to 2000 records, and you wish the additional records to contain bytes that are all zeros. You could use the following program segment:

```
DECLARE C CHARACTER(28);
C = LOW(28);
...
WRITE FILE(F) FROM(C) KEYFROM(2000);
```

The function reference LOW(28) returns a CHARACTER string containing 28 bytes, all of which are zeros. The assignment statement assigns this string to the CHARACTER variable C, and the WRITE statement outputs that variable to the file. If the current size of the file is 1000 records, this WRITE statement adds 1000 new records to the file, and each of the new records contains all zero bytes.

DIRECT ACCESS WITH MIDASPLUS FILES

This section outlines how you use MIDASPLUS files from PL/I. For further examples and details, see the section on PL/I-G in The MIDASPLUS User's Guide.

Prime PL/I uses the attributes KEYED SEQUENTIAL to indicate a MIDASPLUS file. Note that even though you use the keyword SEQUENTIAL for MIDASPLUS, you may nonetheless use direct access operations on these files.

Each record of a MIDASPLUS file has a key. The important characteristics of the keys of MIDASPLUS files are as follows:

- Each record of a MIDASPLUS file has a key that may be an arbitrary CHARACTER string. The length of the key is the same for all records of the file.
- If PL/I creates the MIDASPLUS file, the length of the key is always precisely 32 characters.
- If you create the file by means of a separate utility and you wish to update the file from a PL/I program, the key length may be 32 characters or less.

Internally, PL/I organizes MIDASPLUS files in two parts, an index portion and a data portion. When you write a new record to the MIDASPLUS file, you must tell PL/I both the data value (using the FROM option) and the key (using the KEYFROM option). Each entry in the index portion of the MIDASPLUS file contains both the value of the key and a pointer to the corresponding data record. When you wish to read a record of the file, use the KEY option to specify the key of the record you desire. PL/I looks up the key in the index portion of the file and uses the pointer it finds there to find the data record that you are requesting.

Operations on MIDASPLUS files are like operations on DAM files, with the following differences:

- The expression that you use with the KEYFROM option of the WRITE statement or the KEY option of the REWRITE or READ statement should be CHARACTER, and the length of the string should be equal to or less than the key length for the file. The key must be unique.
- You may use the WRITE statement to insert new records into any point in the file, not just at the end.
- You may use the DELETE statement to delete any record of the file. When you use the DELETE statement, PL/I deletes both the key from the index portion of the file and the data record from the data portion of the file.

The Basic Statements

Let us look at some specific examples of statements you use to access MIDASPLUS files. You can DECLARE a MIDASPLUS file as follows:

```
DECLARE F FILE RECORD KEYED SEQUENTIAL;
```

This declaration uses the attributes KEYED SEQUENTIAL, which signal to PRIME PL/I that F is a MIDASPLUS file.

To create a MIDASPLUS file using the declaration just above, you could use statements like

```
DECLARE KVBLE CHAR(32);
OPEN FILE(F) OUTPUT TITLE('MIDASPLUSFILE');
...
WRITE FILE(F) FROM(S) KEYFROM(KVBLE);
...
CLOSE FILE(F);
```

The OPEN statement specifies the OUTPUT attribute, which means that PL/I creates a new file called MIDASPLUSFILE.

Each WRITE statement of the type shown in the example adds a new record to the file. The FROM and KEYFROM options of the WRITE statement specify what is to be stored in the data portion and the index portion, respectively, of the MIDASPLUSFILE. The data stored in the data record of the file is copied from the storage area occupied by the variable S, used with the FROM option. PL/I takes the 32 characters from the variable KVBLE, and stores that string in an entry in the index portion of the file. That entry also contains a pointer to the data record that we have just written.

The final statement of the program example above is the CLOSE statement. When you execute that statement, the file creation process is complete.

Once the file is created you may update it, in the same program or in a different program, by using the OPEN statement with the UPDATE option:

```
OPEN FILE(F) UPDATE TITLE('MIDASPLUSFILE');
```

After PL/I has executed this OPEN statement, your program may perform any of the following operations:

- Use the statement

```
READ FILE(F) INTO(S) KEY(KVBLE);
```

to have PL/I search the index portion of the MIDASPLUS file for an index entry that specifies a key equal to the KVBLE CHARACTER string, and then read the corresponding data record from the data portion of the file into the storage block occupied by the variable S.

- Use a typical WRITE statement, such as

```
WRITE FILE(F) FROM(S) KEYFROM(KVBLE);
```

to add a new record to the MIDASPLUS file. The string in KVBLE is used as the key to be stored in the index portion of the file, and the data record is taken from the storage area occupied by the variable S.

- To change the data record for an existing key, use

```
REWRITE FILE(F) FROM(S) KEY(KVBLE);
```

This statement is legal only if the file already contains a record with key specified by the string variable KVBLE. PL/I uses the data in the storage block occupied by S to replace the data record for this key. The old data record is lost.

- Finally, use the statement

```
DELETE FILE(F) KEY(KVBLE);
```

to delete the record with the key specified by the string in KVBLE. PL/I deletes both the data record in the data portion of the file and the index entry in the index portion of the file.

Use the WRITE statement to add a new record to the file, and REWRITE to replace an existing record. The REWRITE statement is legal only if a record with the specified key already exists, and the WRITE statement is legal only if no record with the specified key already exists.

Finally, suppose that you have created an updated MIDASPLUS file, and that you wish your program to make a list of all the records in the

file in alphabetical order by key value. (By alphabetical order, we mean the collating sequence order shown in Appendix B.) To do this, your program may execute an OPEN statement with the INPUT option. Then you may execute READ statements like

```
READ FILE(F) INTO(S) KEYTO(KVBLE);
```

which uses a new option, the KEYTO option. Since this statement contains no KEY option, PL/I reads the file in sequential order; more precisely, in alphabetical order by key. The READ statement takes the value of the key from the index portion of the file and stores it in the variable KVBLE. It also takes the corresponding data record from the data portion of the file and stores it in the variable S. You may then print out the value of either the key (in KVBLE) or the data (in S).

The argument in the KEYTO option must be a varying CHARACTER string, while the KEY and KEYFROM arguments may be either varying or nonvarying.

RECORD INPUT/OUTPUT IN LOCATE MODE

To improve the efficiency of your RECORD input/output statements, use locate mode operations, which improve efficiency by reducing the need for PL/I to copy blocks of data between internal buffers. In order to explain how locate mode works, we must first explain how PL/I's internal buffering works.

Most RECORD input/output statements operate through an internal buffer that is usually invisible to the programmer. For example, a statement like

```
READ FILE(F) INTO(S);
```

is executed according to the following steps:

1. PL/I transmits the data record from the input device into an internal buffer that is invisible to the user.
2. PL/I copies the data record from the internal buffer to the storage area for S.

Similarly, when PL/I executes a WRITE statement, it copies the data to an internal buffer that is invisible to the user, and the data is transmitted to the device from that internal buffer.

By using locate mode input/output, you can improve the performance of your input/output operations by avoiding the step of copying the data

between the internal buffer and the storage area. PL/I provides a POINTER value to the internal buffer, so that you can manipulate the data directly in this buffer, rather than in the storage area of your program.

Locate Mode Input: The SET Option

Up to this point, each READ statement that we have seen has had an INTO option to specify the name of a variable in whose storage area the transmitted data is to be stored. You can rewrite such a READ statement, replacing the INTO option with a SET option with the following syntax:

```
SET(pointer-variable)
```

For example, by replacing the INTO option with a SET option, you can change

```
READ FILE(F) INTO(S);
```

to

```
READ FILE(F) SET(P);
```

where P is a variable with the POINTER data type. In this second form of the READ statement, PL/I transmits the data from the device into its internal buffer, but does not copy this record into the storage block occupied by a variable. Instead, PL/I sets P to the address of the record in the internal buffer. You may then access the data directly using the POINTER variable P and an appropriate BASED variable, as discussed in Chapter 7.

Locate Mode Output

Output in locate mode is conceptually a bit more subtle than input in locate mode. The program segment that follows uses non-locate mode WRITE statements.

```

DECLARE 1 REC,
        2 NAME CHAR(20),
        2 AGE BIN FIXED;
DECLARE F FILE RECORD SEQUENTIAL;
...
OPEN FILE(F) OUTPUT;
REC . NAME = 'JONES';
REC . AGE = 23;
WRITE FILE(F) FROM(REC);
REC . NAME = 'SMITH';
REC . AGE = 45;
WRITE FILE(F) FROM(REC);
CLOSE FILE(F);

```

This program segment opens a file, writes two data records to the file, and closes it.

To rewrite this example in locate mode, first obtain a POINTER value to PL/I's internal output buffer by using the LOCATE statement. Next, use the POINTER value with a BASED structure to move data values into the output buffer. Then execute another LOCATE statement to tell PL/I that you have finished filling the first output buffer and are ready for another output buffer.

The following PL/I program segment performs these operations:

```

DECLARE P POINTER,
        1 REC BASED,
        2 NAME CHAR(20),
        2 AGE BIN FIXED;
DECLARE F FILE RECORD SEQUENTIAL;
...
OPEN FILE(F) OUTPUT;
LOCATE REC SET(P);
P->REC . NAME = 'JONES';
P->REC . AGE = 23;
LOCATE REC SET(P);
P->REC . NAME = 'SMITH';
P->REC . AGE = 45;
CLOSE FILE(F);

```

This example illustrates the new form of output statement, the LOCATE statement. The first LOCATE statement allocates an internal buffer and sets the POINTER variable P to its address. The program then uses BASED storage with P to store values in that buffer. The second LOCATE statement in the example has a dual purpose:

- It tells PL/I that you are finished with the first output buffer. As a result, PL/I can transmit the first output buffer to the output device.

- It allocates a new output buffer for the second output record and sets the POINTER variable P to the address of that buffer.

This example has only two LOCATE statements, but if it had more, each additional LOCATE statement would transmit the preceding buffer to the output device and would allocate a new output buffer.

The CLOSE statement, which is the last statement of the example, completes the locate mode output operations by transmitting the last output buffer to the device and closing the file.

The syntax of the LOCATE statement is as follows:

```
LOCATE identifier SET(variable) FILE(identifier) other-options;
```

In this syntax, the identifier must be a BASED variable so that PL/I will know how large the output buffer must be. The variable appearing with the SET option must have the POINTER data type. The other-options are options of the WRITE statement, except that the FROM option may not be used.

FILE ATTRIBUTES, ATTRIBUTE MERGING, AND THE OPEN STATEMENT

Note

This section applies to both STREAM and RECORD input/output.

In a DECLARE or OPEN statement for a file, you specify various file attributes and options, such as RECORD or STREAM, INPUT or OUTPUT or UPDATE, and so forth. This section discusses these attributes.

When you access a file or device, you use file attributes to describe various things about the operation, such as how the file is organized, the formats and size of the records of the file, and the way in which you will access the file. Prime PL/I permits you to specify file attributes and options in three different contexts:

- As attributes in the DECLARE statement that declares the FILE identifier;
- As options of the OPEN statement, which you use to open the file; or
- In the CHARACTER string value used with the TITLE option of the OPEN statement, which opens the file.

Two kinds of attributes are standard and nonstandard attributes. The attributes that you specify by either of the first two methods above

(as attributes in the DECLARE statement or as options in the OPEN statement) are standard, in the sense that these attributes are defined by the ANS PL/I standard and are the same in all implementations of the PL/I language, including those implementations on other computers. The attributes that you specify in the CHARACTER string associated with the TITLE option of the OPEN statements are not defined by the ANS PL/I standard, and they are unique to the Prime implementation of the PL/I language.

Table 12-1 gives all legal file attributes in the Prime PL/I language. This table breaks the attributes down into four groups. The ones to the left are for STREAM input/output, and the ones to the right are for RECORD input/output. Within these two divisions, the table breaks the attributes down according to whether they are standard and may be specified as either attributes in the DECLARE statement or options in the OPEN statement, or nonstandard and must be specified in the CHARACTER string values associated with the TITLE option. The square brackets used with the nonstandard options for RECORD input/output in the table signify that the decimal integer associated with that option need not always be specified.

Table 12-1
File Attributes

	For STREAM Input/Output	For RECORD Input/Output
Standard Attributes in DECLARE or OPEN Statement	STREAM INPUT or OUTPUT PRINT	RECORD INPUT or OUTPUT or UPDATE SEQUENTIAL or DIRECT KEYED
Attributes in TITLE Option of OPEN Statement	-SAM -APPEND -DEVICE	-SAM [n] -DAM [n] -APPEND [n] -DEVICE [n] -RECL n -FUNIT m -NOSIZE -CITLASA

Once a file has been opened, only certain input/output statements are legal for that file, depending upon the attributes that were specified when the file was opened. Table 12-2 lists all those statements that are legal, depending on whether the file was opened for STREAM or RECORD I/O, and depending upon whether the file was opened for INPUT, OUTPUT, or UPDATE.

Table 12-2
Legal Statements

	For STREAM Input/Output	For RECORD Input/Output
INPUT	GET READ (see below)	READ
OUTPUT	PUT WRITE (see below)	WRITE DELETE LOCATE
UPDATE	(Illegal)	READ WRITE REWRITE DELETE LOCATE

You can use nonstandard READ and WRITE statements with STREAM I/O, but you will lose program portability. Refer to Chapter 11 for more information.

Let us now discuss the meanings of some of the other attributes.

The PRINT attribute is permitted only with STREAM OUTPUT file accessing. It specifies that the output file is intended to be printed on a terminal or line printer and is not intended for input to another program. The PRINT attribute is discussed in detail in Chapter 11.

The KEYED, SEQUENTIAL, and DIRECT attributes apply to RECORD input/output only. The KEYED attribute specifies a file organization where each record of the file has a key. For DAM files, the key is a relative record number. For MIDASPLUS files, the key is an arbitrary string of characters, such as a person's name or a social security number.

Use the SEQUENTIAL and DIRECT attributes to tell how you plan to access the file. The SEQUENTIAL attribute means that you plan to access the records of the file in ascending order by key, while the DIRECT attribute means that you plan to use direct access methods, where you will jump around. In the Prime implementation, however, KEYED SEQUENTIAL is the signal for MIDASPLUS files, and direct access operations are permitted even though the SEQUENTIAL attribute is used.

Formats of the TITLE Option CHARACTER String

You may use the TITLE option in the OPEN statements. It has the syntax

```
TITLE(expression)
```

where the expression is any legal PL/I expression. PL/I evaluates this expression and converts it to CHARACTER if it is not already CHARACTER.

The characters in this CHARACTER string must have one of the following forms:

```
name
name -option
name -option [n]
```

In these forms, the name is either a file name or a device name. The option is a nonstandard option, one of those specified below; n is a decimal integer that you use to specify the maximum record size of the file or device in words. If you do not specify n, PL/I uses a default value of 1024.

The full format of the TITLE option is

```
OPEN FILE(f) TITLE('name [{-SAM|-DAM|-DEVICE}] [-APPEND]
                  [[-RECL] n] [-FUNIT m] [-NOSIZE] [-CTLASA]')
```

where name is the pathname or device name, which must be supplied if TITLE is present. If the name begins with @, and no options are supplied, the @ is removed and the name is compared with a list of device names (see Table 12-3). If a match is found, the file is associated with the specified device. If no match is found, default values are assumed for the other options.

The nonstandard options that you specify in the TITLE option are as follows:

- **-SAM [n]:** The file has been or will be organized as a SAM file. You may access it only by sequential access methods, and the records have no keys. The value of n, whether you specify it explicitly or use the default value of 1024, is the maximum number of words in each record. In the WRITE and LOCATE statements that follow the OPEN statement, your program may output records shorter than n words; in this case, PL/I writes only the smaller number of words.

- **-APPEND [n]**: The file is like a SAM file, except that **-APPEND** is legal only for **OUTPUT**, and the new output records are appended to the end of an existing disk file. For **-SAM** with an **OUTPUT** file, an old file with the same name is deleted.
- **-DAM [n]**: The file has a DAM file organization. As explained earlier in this section, you may use **DIRECT** access operations with these files, and the key of an individual record in the file is a decimal integer representing the relative record number of the record within the file. An important difference between SAM files and DAM files is that, for DAM files, in the **WRITE** and **LOCATE** statements that follow the **OPEN** statement, each record transmitted to the file contains precisely n words, whether you specify n explicitly, or use the default value of n = 1024. Therefore, all records in a DAM file have precisely the same size, although they may have different sizes in a SAM file.
- **-DEVICE [n]**: This is like **-SAM**, except that the name is interpreted as a device name rather than as a disk file name.

Table 12-3 lists all the legal device names. If you specify **-DEVICE**, the name must be one of the device names given in this table. If you specify **-SAM**, **-DAM**, or **-APPEND**, the name may be any disk file name, and is interpreted as such. If you specify the name with no additional option, PL/I chooses an option according to the following rules:

1. If the file is open with the **RECORD KEYED SEQUENTIAL** attributes, the name is interpreted as the name of a **MIDASPLUS** disk file.
2. If the file is open with the **RECORD DIRECT** attributes, the option **-DAM** is assumed.
3. If the file is opened with the **STREAM** attribute, or if the file is opened with the **RECORD SEQUENTIAL** attribute, but without **KEYED**, and if the name is one of the valid device names listed in Table 12-3, the option **-DEVICE** is assumed.
4. If the conditions specified in the preceding paragraph hold, except that the name is not listed in Table 12-3, **-SAM** is assumed.

Table 12-3
Device Names

Name	Input/Output/Update	Device
SYSIN	INPUT	Terminal
SYSPRINT	OUTPUT	Terminal
TTY	INPUT/OUTPUT	Terminal
PTR	INPUT	Paper tape reader
PTP	OUTPUT	Paper tape punch
CR	INPUT	Card reader
SER	OUTPUT	Serial printer
MT0-MT7	INPUT/OUTPUT/UPDATE	Mag tape drives 0-7
PR0-PR1	OUTPUT	Line printer 0-1

- **-RECL n:** This option specifies the record length (in bytes) for DIRECT files, or the buffer size (that is, maximum record length) for other file types. The maximum is 131,062 bytes. If **-RECL** is omitted, **n** must immediately follow **-SAM**, **-DAM**, **-DEVICE** or **-APPEND**. The default is **-RECL 2048**.
- **-FUNIT m:** This option specifies the file unit on which the disk file is to be opened, or is already open. If the file is already open, **name** is ignored. If the file is already open and **-APPEND** is not specified, the file is truncated at its current position. If **-FUNIT** is not specified, any available file unit is used.
- **-NOSIZE:** This option specifies that records of a DIRECT file are to be stored in the old format (the data is not preceded by a word indicating the record length). If **-NOSIZE** is not specified, records are stored in RDBIN/WRBIN format (a word indicating record length precedes the data).
- **-CTLASA:** This option specifies that FORTRAN control codes are to be the first character in each line of a file. The characters and their spacing effect are

<u>Print Option</u>	<u>Character Generated</u>	<u>Effect</u>
SKIP(0)	+	overprinting
PAGE	1	advance to next page
SKIP	blank	one line

This option is useful with the `-FTN` command line option of `SPOOL`. This option allows `SKIP(0)` to be used for overstriking. `SKIP(0)` directed to the `TTY` device causes a carriage return without a line feed.

- `-FORMS`: This option specifies that the file is a `FORMS` file. `FORMS` files must be `STREAM` files. See Appendix H for more information.

The above options may be specified in any order, but name must be the first entry in the `TITLE` option. All names and options are mapped to uppercase before processing. Therefore, `@tty -dev` and `@TTY -DEV` are equivalent. The maximum length of the `TITLE` option is 128 characters.

If the `OPEN` statement contains no `TITLE` option, PL/I assumes a default of `TITLE('ident')`, where ident is the name of the `FILE` identifier in the `FILE` option of the `OPEN` statement. For example,

```
OPEN FILE(F) OUTPUT;
```

is the same as

```
OPEN FILE(F) OUTPUT TITLE('F');
```

Explicit and Implicit File Openings

Before PL/I can execute any `GET`, `PUT`, `READ`, `WRITE`, `REWRITE`, `LOCATE`, or `DELETE` statements, PL/I must open the file. Your program may open a file explicitly by executing an `OPEN` statement.

However, if your program executes one of the above input/output statements for a file that is not open, PL/I opens the file implicitly before executing the I/O statement. The implicit opening is equivalent to an explicit `OPEN` statement with certain implied standard attributes that depend upon the type of the statement that is causing the file to be opened implicitly. Table 12-4 lists these implied attributes for each of the different kinds of statements.

Suppose, for example, your program executes the statement

```
GET FILE(F) LIST(X);
```

and the file `F` has not been opened. Then PL/I opens the file implicitly, by simulating execution of the following statement:

```
OPEN FILE(F) STREAM INPUT;
```

Table 12-4
Implied Attributes for I/O Statements

Statement	Implied Attributes
GET	STREAM INPUT
PUT	STREAM OUTPUT
READ	RECORD INPUT*
WRITE	RECORD OUTPUT*
REWRITE	RECORD UPDATE
LOCATE	RECORD OUTPUT
DELETE	RECORD UPDATE

* INPUT and OUTPUT are not implied if UPDATE is a DECLARE attribute.

Attribute Merging and Completion at File Opening

As we have previously stated, you may specify attributes of a file in several different places: as attributes in the DECLARE statement, as options in the OPEN statement, and in the CHARACTER string value of the expression specified with the TITLE option. When your program executes any statement that opens a file, PL/I must form a complete, consistent attribute set for the file. This means that PL/I must gather together all the file attributes specified in the various possible places, must check them for consistency, and must supply additional attributes when the specified attribute set is incomplete. This routine is necessary whether the file is opened by an explicit OPEN statement or by another input/output statement that causes PL/I to execute an implied OPEN statement as described in the preceding section.

PL/I goes through the following steps to supply attributes:

1. PL/I starts with the attributes given in the declaration of the file.
2. If the file opening is caused by an explicit OPEN statement, PL/I merges the attributes in the OPEN statement with the attributes in the declaration.
3. If the file opening is implicit, PL/I merges the attributes implied by the input/output statement type (see the preceding section) with the attributes in the declaration.

4. Certain file attributes imply other file attributes. Table 12-5 lists these attribute implications. If the merged attribute set so far contains any of the attributes in the first column of Table 12-5, PL/I adds the attributes in the corresponding line of the second column to the merged attribute set.
5. If the merged attribute set so far contains neither the `STREAM` attribute nor the `RECORD` attribute, PL/I uses the default attribute of `STREAM`.
6. If the merged attribute set so far does not contain any of the attributes `INPUT`, `OUTPUT`, or `UPDATE`, PL/I uses the default attribute of `INPUT`.
7. If the merged attribute set so far contains `RECORD`, but does not contain either `DIRECT` or `SEQUENTIAL`, PL/I uses the default attribute of `SEQUENTIAL`.
8. If the file being opened has an identified name of `SYSPRINT`, and if the merged attribute set so far contains `STREAM` and `OUTPUT`, PL/I adds `PRINT` to the merged attribute set.

Table 12-5
Implied Attributes for Files

Attribute	Implied Attributes
<code>DIRECT</code>	<code>RECORD KEYED</code>
<code>KEYED</code>	<code>RECORD</code>
<code>PRINT</code>	<code>STREAM OUTPUT</code>
<code>SEQUENTIAL</code>	<code>RECORD</code>
<code>UPDATE</code>	<code>RECORD</code>

PL/I follows the above steps in order to build a complete attribute set. After these steps have been completed, PL/I must do some error checking of several different types.

First, PL/I must check for conflicting attributes. Table 12-6 lists all conflicting file attributes. If the attribute set contains any attributes in the first column along with an attribute on the same line in the second column, there is an attribute conflict, and the file opening fails.

Table 12-6
Conflicting Attributes

Attribute	Conflicting Attributes
INPUT	OUTPUT, UPDATE
OUTPUT	UPDATE
STREAM	RECORD
DIRECT	SEQUENTIAL

If the file opening is being done by an explicit OPEN statement, PL/I performs the following additional checking:

1. If the explicit OPEN statement has a LINESIZE option, it is an error if the attribute set does not contain STREAM OUTPUT.
2. If the OPEN statement has a PAGESIZE or TAB option, it is an error if the attribute set does not include the STREAM OUTPUT PRINT options.

If any of the above error checks fail, the entire file opening fails, and PL/I raises the UNDEFINEDFILE condition.

Attribute Requirements of Input/Output Statements

Before PL/I can execute any input/output statement, the file must be opened with certain attributes. For example, before PL/I can execute a GET statement, the file must be opened with the STREAM INPUT attributes. Each statement type has its own set of required attributes. Table 12-7 lists all of these required attributes. Each statement of the type listed in the first column may not be executed unless the file has been opened, either implicitly or explicitly, with the attributes in the second column. If the file is not open, PL/I opens it implicitly, according to the rules already given. If the file has been opened, but without all the attributes specified by the table, PL/I raises the ERROR condition.

Table 12-7
Required Attributes for I/O Statements

Statement	Required File Attributes
READ	INPUT or UPDATE
READ with IGNORE option	SEQUENTIAL
READ with no KEY option	SEQUENTIAL
READ with KEY or KEYTO option	KEYED
WRITE	either OUTPUT or DIRECT UPDATE
WRITE with KEYFROM option	KEYED
LOCATE	RECORD OUTPUT
LOCATE with KEY option	KEYED
REWRITE	RECORD UPDATE
REWRITE with no KEY option	SEQUENTIAL
REWRITE with KEY option	KEYED
DELETE	RECORD UPDATE
DELETE with no KEY option	SEQUENTIAL
DELETE with KEY option	KEYED
GET	STREAM INPUT
PUT	STREAM OUTPUT

There is an additional requirement. If your program executes certain input/output statements, certain options of those statements are required, depending upon the attributes of the open file. These requirements are listed in Table 12-8.

Table 12-8
Required Attributes for File Options

Statement	If the Open File Has These Attributes	Then the Statement Must Have These Options
REWRITE	DIRECT	KEY
LOCATE	DIRECT	KEY

If any requirements listed above fail, PL/I raises the ERROR condition.

The Effect of the CLOSE Statement on Attributes

When PL/I executes a CLOSE statement, the attributes associated with the FILE identifier revert to those specified in the declaration alone. That is, the attribute set for the file that was created when the file was opened is thrown away, leaving only those attributes in the DECLARE statement for the file. This means that it is possible to reopen the file in the same program with different attributes.

The program segment

```
DECLARE F FILE RECORD;
OPEN FILE(F) OUTPUT;
...
WRITE FILE(F) ...;
...
CLOSE FILE(F);
OPEN FILE(F) INPUT;
...
READ FILE(F) ...;
...
CLOSE FILE(F);
```

illustrates how you can close a file and then reopen it with different attributes. The first OPEN statement opens the file F for RECORD OUTPUT, presumably to create a disk file. When PL/I executes the CLOSE statement, PL/I leaves F only with the RECORD attribute specified in the DECLARE statement. Then, the new OPEN statement opens the same file for RECORD input.

Be aware of a fairly subtle potential error. If you execute an OPEN statement for a file that is already open, PL/I simply ignores the OPEN statement. For example, in the program segment shown above, suppose you accidentally omitted the CLOSE statement from your code. Then when PL/I encountered the second OPEN statement, PL/I would ignore the OPEN statement, because the file was already open, and so the file would remain open with the RECORD OUTPUT attributes obtained from the first OPEN. PL/I would not find any error until your program attempted to execute the READ statement, which would fail because the file was not opened with either the INPUT or UPDATE attributes.

INPUT/OUTPUT ON CONDITIONS AND BUILT-IN FUNCTIONS

Note

This section applies to both `STREAM` and `RECORD` input/output.

When a statement execution fails because of an error, PL/I normally prints an error message and terminates execution of your program. You may, however, use the `ON` statement to specify what action your program should take when an error occurs. `ON` statements are presented in detail in Chapter 13.

The following is a list of all `ON` conditions for input/output operation. In the following list, identifier is the `FILE` identifier specified with the `FILE` option of the input/output statement.

- `UNDEFINEDFILE(identifier)` is raised when either an implicit opening or an explicit `OPEN` statement fails for any reason.
- `ENDFILE(identifier)` is raised when a `GET` or `READ` statement fails because of end of file. It is discussed more fully in Chapter 11.
- `KEY(identifier)` is raised when a keyed operation fails. This can occur when a key specified in either the `KEY` or `KEYFROM` option has the wrong format, when a record with the specified key does not exist in the file for a `READ`, `REWRITE`, or `DELETE` statement, or when a record with the specified key already does exist in the file for a `WRITE` statement.
- `RECORD(identifier)` is raised when the size of the variable specified with the `FROM` or `INTO` option is too large or too small for the file or device being processed by the input/output statement.
- `TRANSMIT(identifier)` is raised when a hardware input/output error occurs during an I/O operation.
- `ENDPAGE(identifier)` is raised when output from a `PUT` statement to a `PRINT` file reaches the bottom of a printer page. It is discussed more fully in Chapter 11.
- `NAME(identifier)` is raised when the input stream for a `GET DATA` operation contains an invalid variable reference to the left of an equal sign.

Since it is possible for a given on-unit to be raised for various conditions and various error situations, PL/I provides several built-in functions for use in on-units to determine precisely what error caused the on-unit to be invoked. These are discussed in Chapters 13 and 14.

The condition-handling built-in functions specifically related to input/output conditions are as follows:

- ONFILE() returns a CHARACTER string containing the name of the FILE identifier for which the input/output condition was raised.
- ONFIELD() returns a CHARACTER string containing the invalid characters in the input stream that caused the NAME condition to be invoked on a GET DATA statement.
- ONKEY() returns a CHARACTER string containing either the invalid value computed from the expression in the KEY or KEYFROM option, or else the key of the record on which the input/output operation fails.

In addition, ONCODE(), while not limited to I/O functions, returns the error code for conditions for which no on-unit was defined.

FILE VARIABLES AND FUNCTIONS THAT RETURN FILE VALUES

Note

This section applies to both STREAM and RECORD input/output.

All the FILE declarations in this and preceding sections have been for identifiers given the FILE CONSTANT attributes. (The CONSTANT attribute, which you normally do not specify yourself, is the default.)

You may also DECLARE an identifier to have the FILE VARIABLE attributes. Such a variable can have as its value any FILE constant. In fact, in an assignment statement, you may assign any FILE constant to a FILE variable.

This is illustrated in the following example:

```

DECLARE(F,G) FILE STREAM OUTPUT;
DECLARE FV FILE VARIABLE;
OPEN FILE(F);
OPEN FILE(G);
...
IF K = 1 THEN FV = F; ELSE FV = G;
...
PUT FILE(FV) LIST(X,Y);

```

In this example, F and G each have the FILE CONSTANT attributes, while FV has the FILE VARIABLE attribute. The OPEN statement opens the two files, F and G. The first use of FV is illustrated in the THEN and ELSE clauses of the IF statement of the example above. As shown, the

value of K determines which of the constants F or G is to be assigned to FV. The PUT statement at the end of the example outputs to either F or G, depending upon which of these has been assigned to FV.

You may also DECLARE an array of FILE VARIABLE values, as in the following:

```
DECLARE(F1, F2, F3, F4, F5) FILE
      RECORD OUTPUT;
DECLARE FA(5) FILE VARIABLE
      INITIAL(F1, F2, F3, F4, F5);

...
DO K = 1 TO 5;
WRITE FILE(FA(K)) FROM(S);
END;
```

In this example, the FILE VARIABLE array has been initialized to the five FILE CONSTANT values, as shown. The WRITE statement is executed five times, once for each of the FILE CONSTANT values.

A user-defined function may return a FILE value if the FILE data type is specified in the RETURNS option, as in

```
FILEFUNC: PROCEDURE(N) RETURNS(FILE);
```

Inside such a procedure, the RETURN statement should specify, as an argument, a constant or variable whose data type is FILE.

13

PL/I Condition Handling

During execution of your PL/I program, it is possible for various conditions to be raised. A condition is something that happens during execution of a statement that alters or prevents the normal execution of that statement. The following kinds of events are conditions:

- An error. For example, the statement

A = B/C;

would not execute normally if C were 0. This is called a ZERODIVIDE error. Many conditions are errors.

- End of file. When a GET or READ statement fails because of end of file, the GET or READ statement cannot execute normally. Even though, philosophically speaking, such a condition is not really an error since end of file must occur sooner or later, nonetheless PL/I treats this situation as if it were an error. This is called an ENDFILE condition.
- Termination of program. Termination of your program for any reason (including normal completion) is a condition.

The following sections show how to use the ON statement to specify what action your program should take whenever a condition such as those just described is raised.

THE ON STATEMENT

The PL/I language specifies a standard system action that PL/I takes whenever a condition is raised. In most (but not all) situations where a condition is raised, the standard system action that PL/I takes is to print an error message and terminate execution of your program.

If you wish PL/I to take an action other than the standard system action, use the ON statement to specify exactly what action PL/I should take.

Although the ON statement has several different forms, the most common form is the following:

```
ON condition-name on-unit;
```

where condition-name specifies the condition for which you wish to specify alternate action; the full list of these condition names is given later in this chapter. The on-unit is the alternate action that PL/I should take instead of the standard system action.

The simplest form of on-unit is a single statement that you wish PL/I to execute when the condition occurs. For example, the statement

```
ON ZERODIVIDE CALL REVALUE;
```

specifies that, in any arithmetic computation, if a condition is raised because of division by zero, PL/I should execute the statement CALL REVALUE;. Frequently the single statement is a GO TO statement. An on-unit that is a single statement may not be an IF, DO, or ON statement.

Alternatively, if you wish the action specified by your on-unit to contain several statements, use a BEGIN/END block. For example, consider the following ON statement:

```
ON OVERFLOW BEGIN;  
  DECLARE X FLOAT;  
  X = A - B + C;  
  IF X < 0 THEN GO TO ERRA;  
    ELSE GO TO ERRB;  
END;
```

This ON statement specifies the OVERFLOW condition, and so this statement tells PL/I what alternate action you wish it to take when floating-point overflow occurs during computation of an expression. Notice that PL/I does not execute the statements between the BEGIN and END right away. Instead, PL/I skips over these statements for the time

being, but when the OVERFLOW condition occurs, PL/I goes back and executes these statements.

When a condition is signalled, each block activation beginning with the current block activation is examined to see if it has an established on-unit for the condition. If it does not, the previous block activation is examined, and so on, until an on-unit for the condition is found. If no on-unit exists, the standard system action is invoked. The system action for KEY or ENDFILE signals the ERROR condition. The system action for ENDPAGE puts a new page. The standard system action for ERROR writes an error message and terminates program execution.

The consequence of this mechanism is that a block may establish its own on-unit for a condition or may let its caller's on-unit handle the condition. Any on-unit established by a block is cancelled when the block returns to its caller or is otherwise terminated.

Other Formats of the ON Statement

For any form of the ON statement, you may specify more than one condition name by giving a list of condition names separated by commas. This applies both to ON statements that specify on-units, and to ON statements with the SYSTEM option. For example,

```
ON UNDERFLOW, OVERFLOW GO TO COMPERR;
```

specifies that on either the UNDERFLOW or OVERFLOW condition, control should pass to the statement with label COMPERR.

The On-unit as a Block

An on-unit may be a single statement, or it may be a group of statements beginning with BEGIN and ending with END. However, even when the on-unit is a single statement, there are implied BEGIN and END statements, because PL/I inserts these statements before and after the on-units, respectively.

In the ON statement

```
ON ENDFILE(SYSIN) FLAG = 1;
```

the on-unit is a single statement, FLAG = 1. Since the on-unit is a single statement, PL/I inserts a BEGIN statement before the on-unit and an END statement after the on-unit, so that the on-unit is really a group of statements beginning with BEGIN and ending with END.

Therefore, the above ON statement is equivalent to

```
ON ENDFILE(SYSIN) BEGIN;  
    FLAG = 1;  
END;
```

In both cases, the on-unit is a block, but in the first case, the BEGIN and END statements are implied.

Normal and Abnormal Termination of the On-unit

Like any other block in your program, an on-unit may terminate normally or abnormally. An on-unit terminates normally if your program executes the END statement of the on-unit block. It terminates abnormally if your program executes a GO TO statement that transfers out of the on-unit.

Therefore, the on-unit specified by the statement

```
ON ENDFILE(SYSIN) FLAG = 1;
```

would terminate normally if invoked, because it is a simple assignment statement. An on-unit specified by

```
ON ENDFILE(SYSIN) CALL DONE;
```

would also terminate normally, since it invokes a procedure. However, the on-unit in the statement

```
ON ENDFILE(SYSIN) GO TO EOF;
```

would terminate abnormally, since control would be transferred to a label, EOF, that is not within the on-unit.

Examples of ON ENDFILE

The most commonly used ON condition name is ENDFILE. An ENDFILE condition is raised when a GET or READ statement fails because end of file was reached on the file being used for input.

The following program segment illustrates how to use GET LIST to input data values and stop at end of file.

```

ON ENDFILE(SYSIN) GO TO NEXT;
  DO WHILE('1'B);
  GET LIST(X);
  PUT LIST(SQRT(X));
  END;
NEXT: ...

```

In this example the ON statement specifies that when end of file occurs on file SYSIN, control should transfer to the statement with label NEXT. (This would be an abnormal termination of the on-unit.) The DO loop specifies an infinite loop, but the loop terminates when the GET statement fails because of end of file. At that point, the on-unit GO TO NEXT is executed. The statement

```
GET LIST(X);
```

is equivalent to the statement

```
GET FILE(SYSIN) LIST(X);
```

because SYSIN is the default file name when your GET statement does not have a FILE or STRING option. That is why SYSIN is the file name used in the ON ENDFILE statement.

The following example illustrates ON ENDFILE somewhat differently:

```

DECLARE C CHARACTER(80);
DECLARE CARDIN FILE;
...
ON ENDFILE(CARDIN) EOF = 1;
EOF = 0;
READ FILE(CARDIN) INTO(C);
  DO WHILE(EOF = 0);
  PUT SKIP LIST(C);
  READ FILE(CARDIN) INTO(C);
  END;

```

In the previous example, the on-unit was a GO TO statement, and so the on-unit would terminate abnormally when invoked. In this new example, the on-unit is a simple assignment statement, and so when the on-unit is invoked it terminates normally rather than abnormally. This has the advantage of making the on-unit more acceptable to those programmers who follow strict structured programming rules. Another difference in this example is that the input statement is a READ statement rather than a GET statement.

The on-unit is the statement `EOF = 1`. The DO loop executes as long as EOF equals 0. This is true until the on-unit is invoked, and that happens when one of the READ statements fails because of end of file. When such an end of file occurs, PL/I invokes the on-unit after setting EOF to 1, the on-unit terminates normally, and PL/I continues execution with the statement following the READ statement that caused the error to occur.

Examples of ON CONVERSION

PL/I raises the CONVERSION condition whenever any conversion from CHARACTER or pictured-character fails because there is an invalid character in the CHARACTER string that is the source of the conversion. The most common example of this occurs when a GET LIST statement fails because the input stream contains an invalid character.

For example, consider the statements

```
DECLARE VAL FLOAT;  
GET LIST(VAL);
```

Suppose that the GET statement executed, and PL/I found the following characters in the input stream:

```
23V74T;
```

The GET statement would fail because of the characters V and T in the input stream. These characters would cause the conversion of '23V74T' to FLOAT to fail, and PL/I would raise the CONVERSION condition.

Your program can handle errors of this type by establishing a CONVERSION on-unit. In this on-unit, you may use the ONSOURCE and ONCHAR built-in functions to determine precisely what error occurred. ONSOURCE() returns the CHARACTER string for which the conversion failed, and ONCHAR() returns the first invalid character. Consider the following example:

```
DECLARE VAL FLOAT;  
ON CONVERSION BEGIN;  
  PUT SKIP EDIT('INVALID CHAR', ONCHAR(),  
    'IN INPUT STREAM', ONSOURCE(), (A));  
  GO TO BADCHAR;  
  END;  
GET LIST(VAL);  
...  
BADCHAR: ...
```

This example specifies that if a CONVERSION error occurs, PL/I should execute the PUT and GO TO statements that are in the on-unit. Suppose that the GET statement executed, and the input stream were

```
23V74T;
```

Then the CONVERSION condition would be raised, and PL/I would invoke the on-unit. The PUT statement in the on-unit would print the following:

```
INVALID CHAR V IN INPUT STREAM 23V74T
```

This PUT STATEMENT uses the ONCHAR and ONSOURCE built-in functions. After executing the PUT statement, PL/I would transfer control to the statement with label BADCHAR, thus terminating the on-unit abnormally.

There is a more complex method for handling CONVERSION errors: use the on-unit specified with the ON CONVERSION statement to repair the CHARACTER string for which the conversion failed. The technique is to assign new CHARACTER string values to ONCHAR or ONSOURCE, and then to terminate the on-unit normally, which causes PL/I to reattempt the conversion. When you use ONCHAR or ONSOURCE in this way, they are called pseudovariables, rather than built-in functions. The method is illustrated as follows:

```
DECLARE VAL FLOAT;
ON CONVERSION BEGIN;
    ONSOURCE() = '0';
END;
GET LIST(VAL);
```

If there is any bad input to the GET statement, PL/I invokes the on-unit specified for ON CONVERSION. This on-unit assigns the string '0' to ONSOURCE as a pseudovvariable. Then when PL/I executes the END statement of the on-unit (so that the on-unit terminates normally), control returns to the GET statement at the point where the conversion error occurred, and PL/I reattempts the conversion with the string value assigned to the ONSOURCE pseudovvariable. The result is that VAL is given the value 0.

In the preceding example, ONSOURCE was used as a pseudovvariable in the CONVERSION on-unit, with the effect of changing the entire string that caused the conversion error. If you wish to change only the one invalid character that caused the CONVERSION error, you may assign a value to ONCHAR, used as a pseudovvariable. PL/I then changes the erroneous string by changing the first invalid character in that string to the one that you assign to ONCHAR. Then, as before, if you terminate the CONVERSION on-unit normally, PL/I returns to the

PL/I Reference Guide

statement that gave rise to the CONVERSION error and reattempts the conversion with the modified input string. If the conversion fails again, PL/I raises the CONVERSION condition again, and the on-unit can assign a value to ONCHAR again, this time presumably changing the next invalid character in the input string.

Consider the following example:

```
DECLARE VAL FLOAT;  
ON CONVERSION BEGIN;  
    ONCHAR() = '0';  
    END;  
GET LIST(VAL);
```

Suppose that the input stream to this program segment is

23V74T

This input stream contains two invalid characters, V and T. Therefore, the GET statement fails, and PL/I invokes the CONVERSION on-unit. The on-unit assigns the character '0' to the ONCHAR() pseudovvariable, which has the effect of changing the first invalid character in the input stream to '0'. Therefore, the new conversion source string is

23074T

When PL/I executes the END statement of the on-unit so that the on-unit terminates normally, PL/I returns to the GET statement and reattempts the conversion.

The conversion fails again, because there is a second invalid character, T, and so the on-unit is invoked a second time. This time the assignment to ONCHAR changes the T, and so the new conversion source string is

230740

When the on-unit terminates normally this time, PL/I returns to the GET statement and reattempts the conversion, which finally succeeds. Therefore, VAL is given the value 230740.

The very astute reader will realize that the preceding example is very dangerous, since it can possibly result in an infinite loop. The reason for this is that it is possible for the invalid character in a conversion to be a 0, and so assigning the value '0' to the ONCHAR pseudovvariable would have no effect on the string being converted.

Therefore the new conversion attempt would always fail, and so PL/I would go into an infinite loop attempting the conversion, failing, invoking the on-unit, and reattempting the conversion. An example of some input stream characters that would in fact give rise to such a situation is the following:

```
'0
```

If these three characters were in the input stream, the first two characters would be valid, and the first invalid character would be the 0. So the statement

```
ONCHAR() = '0';
```

would make no change to the invalid string of characters taken from the input stream.

This example illustrates an important fact: use of ONCHAR and ONSOURCE as pseudovariabes can be very tricky and can lead to infinite loops unless great care is taken. For this reason, it is recommended that you avoid using these pseudovariabes altogether, and that you terminate your CONVERSION on-units abnormally, executing a GO TO statement out of any CONVERSION on-unit.

Examples of ON ERROR

ERROR is a condition keyword that is a general catch-all for any type of error situation. PL/I raises the ERROR condition in any of the following situations:

- If a program error occurs for which PL/I has no special condition keyword. For example, if you execute a GET statement on a file that has previously been opened for OUTPUT, PL/I directly raises the ERROR condition, with an appropriate ONCODE message indicating the problem.
- If a program error occurs for which PL/I does have a special condition keyword, but your program has no established on-unit for that condition. For example, if a GET statement fails because of end of file on file SYSIN, but your program has not established any on-unit for ENDFILE(SYSIN), PL/I raises the ERROR condition. In fact, this can be restated as follows: the standard system action for the ENDFILE condition is to raise the ERROR condition.

PL/I Reference Guide

- If an on-unit terminates normally, the action taken for certain conditions whose on-units terminate normally is to raise the ERROR condition. For example, if an OVERFLOW on-unit terminates normally, PL/I raises the ERROR condition.

Therefore, as you can see, use of ON ERROR provides a very general method for handling many different kinds of errors. To understand some of the full power of ON ERROR, consider the following program examples:

```
    ...  
    ... Section A  
    ...  
    ON ERROR GO TO LC;  
    ...  
    ... Section B  
    ...  
LC:  ON ERROR GO TO LD;  
    ...  
    ... Section C  
    ...  
LD:  ...  
    ... Section D  
    ...
```

This program has four sections, which are shown as A, B, C, and D. Assuming that the four sections execute in order, and that there is no jumping around among the sections, note what happens if an error occurs in any of these four sections:

- If the ERROR condition is raised in Section A before any ON statement is executed, PL/I takes the standard system action, which is to print an error message and terminate execution of the program.
- Now, suppose that the ERROR condition is raised in Section B, after the first ON ERROR statement is executed. Then PL/I invokes the established on-unit for the ERROR condition, which is

```
    GO TO LC;
```

and so control passes to the statement with label LC. This means that the rest of Section B, after the error occurs, is skipped.

- Now, suppose that control reaches label LC. This might have happened either because of the normal flow of execution out of Section B or because of an error during execution of Section B that caused PL/I to invoke the established on-unit and execute a GO TO to label LC. When PL/I executes the second ON ERROR

statement, the on-unit specified there replaces the on-unit established by the previous ON statement. Therefore, if an error occurs in Section C, PL/I invokes the new established on-unit, which is

```
GO TO LD;
```

and so control passes to the statement with label LD.

- Finally, suppose that control reaches label LD, either through normal flow of control out of Section C, or because of an error in Section C. If an error occurs in Section D, then, since the on-unit

```
GO TO LD;
```

is still established, control returns to the statement with label LD. This will probably result in re-execution of the statement that gave rise to the error, and so an infinite loop may result.

The main point to understand from the above example is that the ON statement is an executable statement, and has no effect until it is executed. (This is unlike the DECLARE statement, which is declarative and is not executed.) Furthermore, when PL/I executes a second ON statement for a given condition, the effect of the first ON statement is wiped out. (This is untrue if the ON statements are in different program blocks.)

The above example also shows the very important point that use of ON ERROR can be very risky. In the example, use of ON ERROR could have led to an infinite loop. The general reason that ON ERROR is so risky is that the ERROR condition can be raised for any sort of error, including many types of errors that you cannot anticipate at the time you write your program. For this reason, use of ON ERROR should be avoided.

In the above example, it is possible to use

```
ON ERROR SYSTEM;
```

PL/I Reference Guide

to avoid an infinite loop. The following is the revised program segment:

```
    ...
    ... Section A
    ...
    ON ERROR GO TO LC;
    ...
    ... Section B
    ...
LC:  ON ERROR GO TO LD;
    ...
    ... Section C
    ...
LD:  ON ERROR SYSTEM;
    ...
    ... Section D
    ...
```

In this revised example, the `SYSTEM` option tells PL/I to take the standard system action when the `ERROR` condition is raised. Therefore, if an error occurs in Section D, PL/I does the same as in Section A: it prints an error message and terminates execution of the program.

Another example of a dangerous use of `ON ERROR` is as follows:

```
ON ERROR PUT DATA;
```

The programmer who codes this PL/I statement intends that PL/I is to dump all program variables in case of any program error that raises the `ERROR` condition. The reason that this statement is dangerous is that `PUT DATA` could itself cause the `ERROR` condition to be raised, and so `ERROR` would be raised again, resulting in an infinite loop. An alternative, and considerably safer, method of doing the same thing is

```
ON ERROR BEGIN;
  ON ERROR SYSTEM;
  PUT DATA;
END;
```

With this on-unit, if an error occurs in the `PUT DATA` statement, the standard system action is taken, and the program terminates.

`ON ERROR` overrides any conditions previously established. Therefore, if you wish to test for other conditions, they should be listed after `ON ERROR`.

ENABLING CONDITIONS WITH CONDITION PREFIXES

The last statement of the program segment

```

DECLARE A(100);
...
K = 101;
...
A(K) = 0;

```

is invalid, because $K = 101$, and so the subscript value is out of range.

PL/I, however, takes no special action, and the results of executing the final assignment statement are unpredictable. The reason for this is that, unless you specify otherwise, PL/I does not monitor subscript errors. Therefore, even though the value of K is 101, PL/I executes

```
A(K) = 0;
```

as if it were a valid statement. Since the array A contains only 100 words, PL/I stores the value 0 in the word following the end of the array A , the word where $A(101)$ would be if it existed. This means that the contents of this word, which might be another variable, or might be an important system pointer, are destroyed, with results that are totally unpredictable.

Whether or not PL/I monitors subscript errors of this kind is under your control. This section discusses these kinds of errors. If PL/I monitors the error, the error condition is said to be enabled; otherwise, the condition is said to be disabled.

Some conditions may be enabled or disabled by means of condition prefixes of the form

(condition-name):

The resulting enablement or disablement has a scope that may be one line, one block, or a whole procedure, depending on the position of the condition prefix. The following sections give examples of enablement, disablement, and scope.

A General Example: Enabling the SUBSCRIPTRANGE Condition

SUBSCRIPTRANGE (abbreviation SUBRG) is the PL/I keyword for the condition corresponding to an illegal subscript reference of the kind

illustrated above. PL/I does not monitor subscript errors unless you direct it to. In PL/I terminology, this is expressed by saying that SUBSCRIPTRANGE is disabled by default. You may enable the SUBSCRIPTRANGE condition by using condition prefixes in the manner described below.

Usually, you enable SUBSCRIPTRANGE for your entire main program or for an entire external procedure. Do this by using a condition prefix before the label of the PROCEDURE statement. Consider the following:

```
(SUBRG):  UPD:  PROC OPTIONS(MAIN);
          ...
          ...
          END UPD;
```

The condition prefix for the PROCEDURE statement is (SUBRG). By placing it prior to the label of the PROCEDURE statement, in the manner shown above, you are specifying that the PL/I compiler is to generate code to check for possible errors in any subscript reference that appears within this main procedure. Similarly, you may use a condition prefix prior to the label of any external procedure to cause PL/I to generate code to monitor subscript errors throughout that procedure.

On the other hand, you may enable the SUBSCRIPTRANGE condition for a single statement only. Do this by using a condition prefix on that single statement. Consider the following, for example:

```
(SUBSCRIPTRANGE):  MAT(I, J) = MAT(I, K) * MAT(K, J);
```

The condition prefix on this assignment statement causes SUBSCRIPTRANGE to be enabled for the assignment statement only. This means that PL/I generates extra code to check that each of the three subscript lists appearing in this assignment statement is valid.

Once you have enabled the SUBSCRIPTRANGE condition, use the ON SUBSCRIPTRANGE statement to establish an on-unit for SUBSCRIPTRANGE, in order to specify what action PL/I should take in case it discovers an error in a subscript reference. For example, consider the following:

```
ON SUBSCRIPTRANGE BEGIN;
  PUT SKIP LIST('SUBSCRIPT ERROR FOR ARRAY A');
  GO TO RESTART;
  END;
(SUBRG):  A(J) = 4 * A(K);
```

SUBSCRIPTRANGE is enabled for the assignment statement that ends this example, and if a subscript error occurs in that assignment statement, the specified on-unit is invoked. If no on-unit had been established, and a subscript error were detected, then PL/I would take the standard system action, which is to print an error message and raise the ERROR condition.

Note that there is no point in having an ON SUBSCRIPTRANGE condition unless you use a condition prefix to enable SUBSCRIPTRANGE. The reason is that PL/I never invokes a SUBSCRIPTRANGE on-unit unless SUBSCRIPTRANGE is enabled.

A General Example: Disabling the CONVERSION Condition

As was discussed earlier in this chapter, PL/I raises the CONVERSION condition when a conversion from a CHARACTER string fails because of an invalid character in the string.

The CONVERSION condition is one of those conditions that are enabled by default, but that you may disable by means of condition prefixes. This means that PL/I checks for invalid characters in source strings unless you specify otherwise. Disable the CONVERSION condition by specifying NOCONVERSION in a condition prefix, as in the following example:

```
(NOCONV): CHR: PROC OPTIONS(MAIN);
           DECLARE X FLOAT;
           DECLARE C CHARACTER(200) VARYING;
           ...
           X = C;
           ...
           END CHR;
```

In this example, the condition prefix is (NOCONV), which precedes the label of the PROCEDURE statement. As a result, CONVERSION is disabled for the entire procedure. Therefore, in the statement X = C, if the string C contains an invalid character, then PL/I misinterprets the invalid character without signaling any error condition, and the result of the conversion is unpredictable.

Enablement/Disablement States of Conditions

Most PL/I conditions are enabled at all times, and you may not disable them. However, certain conditions may be enabled or disabled, depending upon what condition prefixes you use. There are two groups of such conditions:

- Those conditions that are enabled by default and may be disabled. These conditions are CONVERSION, FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, ZERODIVIDE, and STRINGSIZE.
- Those conditions that are disabled by default and may be enabled. Those conditions are SIZE, SUBSCRIPTRANGE, and STRINGRANGE.

All other conditions are enabled by default and cannot be disabled. Enable a condition for a statement or for a block by using the keyword for that condition in the condition prefix for the statement or block. Disable the condition by using NO together with the keyword in the condition prefix in the same way. For example,

```
(SIZE, NOOVERFLOW): M = X + 1;
```

is an assignment statement with the SIZE condition enabled and the OVERFLOW condition disabled.

Similarly, if a BEGIN block begins with the statement

```
(NOOVERFLOW, NOUNDERFLOW, NOZERODIVIDE):  
BL: BEGIN;
```

the conditions OVERFLOW, UNDERFLOW, and ZERODIVIDE are disabled for all statements within the block BL.

When you apply a condition prefix to a PL/I statement, the scope of the condition prefix is the portion of your PL/I program to which that condition prefix applies. The rules for the scope of a condition prefix are as follows:

- The scope of a condition prefix applied to a BEGIN or PROCEDURE statement is the entire block.
- The scope of a condition prefix applied to a DO statement is just the DO statement itself. The condition prefix does not apply to the entire DO group.

- The scope of a condition prefix applied to an IF statement is only the expression immediately following the IF keyword. Thus, the condition prefix applies only to the computation of this logical expression. The condition prefix does not apply to the THEN or ELSE clauses of the IF statement.
- The scope of a condition prefix applied to an ON statement does not include the on-unit.

In all other cases, the condition prefix applies to the entire statement.

If you use a condition prefix to enable or disable some conditions for an entire block, you may override that condition prefix selectively within the block by using condition prefixes within the block. Consider the following example:

```
(NOOVF, SIZE): VARD: PROC OPTIONS(MAIN);
  ...
(NOSIZE):      M = M + 1;
  ...
(OVERFLOW):   BEGIN;
  ...
              END;
              END VARD;
```

The SIZE condition is enabled for the entire main program, except for the assignment statement, for which SIZE is disabled. The OVERFLOW condition is disabled for the whole main program, except within the BEGIN/END block.

Enabling Conditions for Debugging

During the debugging phase of program development, it is most useful for you if PL/I monitors as many errors as possible, even if that means sacrificing space and execution time. Once your program is debugged, and you are ready to put it into production, you usually wish to get maximum execution speed by suppressing some of the error checking.

In practical terms, this translates to the following advice: enable the SIZE, SUBSCRIPTRANGE, and STRINGRANGE conditions during the debugging phase of your program, and leave these conditions disabled after your debugging is complete. For example, if you have a main program called SCHED, you might code it according to the skeleton that follows.

```
(SIZE, STRINGRANGE, SUBSCRIPTRANGE):  
SCHED: PROC OPTIONS(MAIN);  
    ...  
END SCHED;
```

The three conditions are enabled by the condition prefix on the line preceding the PROCEDURE statement, and so the compiler generates code to check for these errors. When you have finished debugging, delete the first line of the program (containing only the condition prefix) and recompile the program, and no code is generated to check for these errors. You will note when you do this that the program with no error checking is smaller and executes faster.

THE REVERT STATEMENT

As has been stated already, when your program executes an ON statement for a condition and that ON statement contains an on-unit, the action specified by the on-unit is said to be established for that condition. If the ON statement has the SYSTEM option rather than an on-unit, the ON statement establishes the standard system action for that condition.

In either case, the action established by the ON statement remains established until one of the following happens:

- If the block in which the ON statement executed terminates, the effect of the ON statement is terminated.
- If the program executes another ON statement for the same condition, the action established by the new ON statement replaces the action specified by the previous ON statement for the same condition.
- If the program executes a REVERT statement for the same condition, within the same block in which the ON statement executed, the effect of the ON statement is wiped out. This is described below.

To understand one important implication of these rules, suppose that block A executes an ON statement and then invokes block B. The action established by the ON statement in block A remains established in block B.

But now suppose that block B executes another ON statement for the same condition. Then the new ON statement establishes a new action to be taken for that condition. However, when block B terminates, and control returns to block A, the effect of the second ON statement is wiped out, and the established action reverts to the action established by the ON statement executed in block A.

This can be restated as follows: if an ON statement establishes an action for a condition, then when the block containing the ON statement

terminates, the effect of the ON statement is also terminated, and the established action for that condition reverts to what it was before the block was invoked.

Use the REVERT statement to do the same thing without terminating the block. Consider the following statement:

```
REVERT condition-name;
```

When PL/I executes this statement, the effect of any ON statement, for the same condition, executed in the same block is wiped out, and the action established for this condition reverts to what it was before the block was entered.

You may specify more than one condition in the REVERT statement. The syntax is

```
REVERT condition-name, condition-name, ...;
```

The established action for each specified condition reverts to what it was before the current block was invoked.

THE SIGNAL STATEMENT

The SIGNAL statement allows you to invoke an on-unit artificially, without actually having the error for the specified condition occur. The statement is structured as follows:

```
SIGNAL condition-name;
SIGNAL CONDITION(file);
SIGNAL CONDITION(user-condition name);
```

When PL/I executes this statement, PL/I raises the specified condition. If the established action for the condition is the standard system action, PL/I prints a message and continues execution of the program with the statement following the SIGNAL statement.

But if the established action is an on-unit, PL/I invokes the on-unit. If the on-unit terminates normally (that is, if your program executes the END statement of the on-unit and does not execute a GO TO statement that transfers out of the on-unit), then PL/I continues execution with the statement following the SIGNAL statement.

The following are the three common uses of the SIGNAL statement:

- To debug an on-unit that cannot easily be invoked by actually having the specified error occur. For example, a TRANSMIT error occurs very rarely and must usually be invoked artificially in order to debug an on-unit.
- To raise the CONDITION condition. This is described below.
- To raise the ENDPAGE condition prior to doing any output to the file, in order to have a page heading at the top of the first page of your output. This is described in Chapter 11.

THE CONDITION CONDITION

The CONDITION condition is a user-defined condition that can be raised only by the SIGNAL statement. For example, you may execute

```
ON CONDITION(BADVALUE) GO TO HANDLE;
```

to establish an on-unit for the user-defined condition, CONDITION(BADVALUE). The only way that this on-unit can be invoked is if your program executes the statement

```
SIGNAL CONDITION(BADVALUE);
```

The identifier BADVALUE, by its appearance with the CONDITION condition, is contextually declared to have the CONDITION attribute. If you wish, you may make this declaration explicit with

```
DECLARE BADVALUE CONDITION;
```

In either event, the identifier BADVALUE may not also be used as a variable or a named constant in your program, since there would be conflicting declarations.

THE SNAP OPTION

To print a partial dump of memory status at the point where an error occurs, use the SNAP option in this format:

```
ON condition SNAP statements
```

The SNAP option causes a trace of stack frame information, useful to those familiar with Prime system architecture, to be displayed on the screen. It may be sent to a file with the COMOUTPUT command, which is explained in the Prime User's Guide in the chapter on command files and phantoms. Sample snap dumps from ON ERROR for an attempt to open a nonexistent file and from ON ZERODIVIDE are reproduced in Figures 13-1 and 13-2.

OK, seg restrictions

UNDEFINEDFILE(FILE1) raised in APPE at 4001(3)/1261

Snap option stack trace for condition "ERROR" follows. (raise_)
Backward trace of stack from frame 2 at 4001(3)/37546.

STACK SEGMENT IS 4001.

(2) 037546: CONDITION FRAME for "ERROR"; returns to 13(3)/45536.
Condition raised at 4001(3)/24074; LB= 4002(0)/4420, Keys= 014040
Return by on-unit is not permitted.

.
.
.

(8) 031676: Owner= APPE (LB= 4002(0)/177400).
Called from 4000(3)/55573; returns to 4000(3)/55575.
Onunit for "ERROR" is 4002(3)/35.

STACK SEGMENT IS 4000.

(9) 150062: Owner= (LB= 4000(0)/55260).
Called from 4000(3)/1702; returns to 4000(3)/1704.
Proceed to this activation is prohibited.

.
.
.

(23) 000352: Owner= (LB= 13(0)/107640).
Called from 1(0)/110224; returns to 1(0)/6003.
Error: condition "ILLEGAL_ONUNIT_RETURN\$" raised at 13(3)/34456.
ER!

Snap Dump From Attempt to Open a Nonexistent File
Figure 13-1

OK, seg restrictions

Snap option stack trace for condition "ZERODIVIDE" follows. (raise_)
Backward trace of stack from frame 2 at 6002(3)/10412.

STACK SEGMENT IS 6002.

(2) 010412: CONDITION FRAME for "ZERODIVIDE"; returns to 13(3)/42672.
Condition raised at 4001(3)/30270; LB= 4002(0)/17672, Keys= 015140
Return by on-unit is not permitted.

·
·

(9) 150062: Owner= (LB= 4000(0)/55260).
Called from 4000(3)/1702; returns to 4000(3)/1704.
Proceed to this activation is prohibited.

(10) 150012: Owner= (LB= 4000(0)/5074).
Called from 4000(3)/1100; returns to 4000(3)/1102.
Onunit for "CLEANUP\$" is 4000(3)/56370.

(11) 150000: Owner= (LB= 4000(0)/5074).
Called from 0(0)/177776; returns to 0(0)/0.

STACK SEGMENT IS 6002.

(12) 007160: Owner= (LB= 13(3)/21530).
Called from 13(3)/6671; returns to 13(3)/6711.
Onunit for "CLEANUP\$" is 13(3)/22215.

·
·

(35) 000352: Owner= (LB= 13(0)/107640).
Called from 1(0)/110224; returns to 1(0)/6003.
Error: condition "ILLEGAL_ONUNIT_RETURNS" raised at 13(3)/34456.
ER!

Snap Dump From ON ZERODIVIDE SNAP
Figure 13-2

LIST OF CONDITIONS

In this section, the names of all the conditions that may be used in the ON statement are listed. Under the Description heading the circumstances under which PL/I raises the condition are explained. Under the Enablement Status heading, there is a definition of whether the condition is enabled or disabled by default, and, if enabled, whether you may disable it. Under the Disabled Result heading, there is an explanation of what happens if the error occurs, but the condition is disabled. The Standard System Action is also described. It tells what PL/I does if there is no established on-unit for the condition. Under the Normal Termination Action heading, there is a description of how PL/I handles the situation where an on-unit for the condition terminates normally (without a GO TO statement that transfers control out of the on-unit.)

Figure 13-3 shows how to interpret the descriptions of each condition name by showing precisely how PL/I handles an error when one occurs.

The top two rows of the figure indicate what happens if there is a condition keyword for the type of error. If the condition is disabled, the result is as specified by disabled result. Otherwise, PL/I may take one of the actions specified by either standard system action or normal termination action, as shown in the figure. The dotted lines in the figure indicate that for many conditions, the standard system action and the normal termination action are to raise the ERROR condition. For some conditions, the normal termination action is to return to the point where the error occurred, and continue execution from there. You must refer to the condition list to learn the precise actions taken for a given condition.

The last two lines of the figure indicate how the ERROR and FINISH conditions are handled. The ERROR condition is a general catch-all condition that can be raised for any type of error, and the FINISH condition is raised whenever your program is about to terminate for any reason, whether because of a normal termination or because of an ERROR. The standard system action and the normal termination action for the ERROR condition are to raise the FINISH condition. The standard system action and the normal termination action for the FINISH condition are to terminate the program.

All the information given for a condition is valid only for the case where the condition is raised as the result of a program error. If the condition is raised as the result of a SIGNAL statement, the standard system action and normal return action are to continue execution with the statements following the SIGNAL statement.

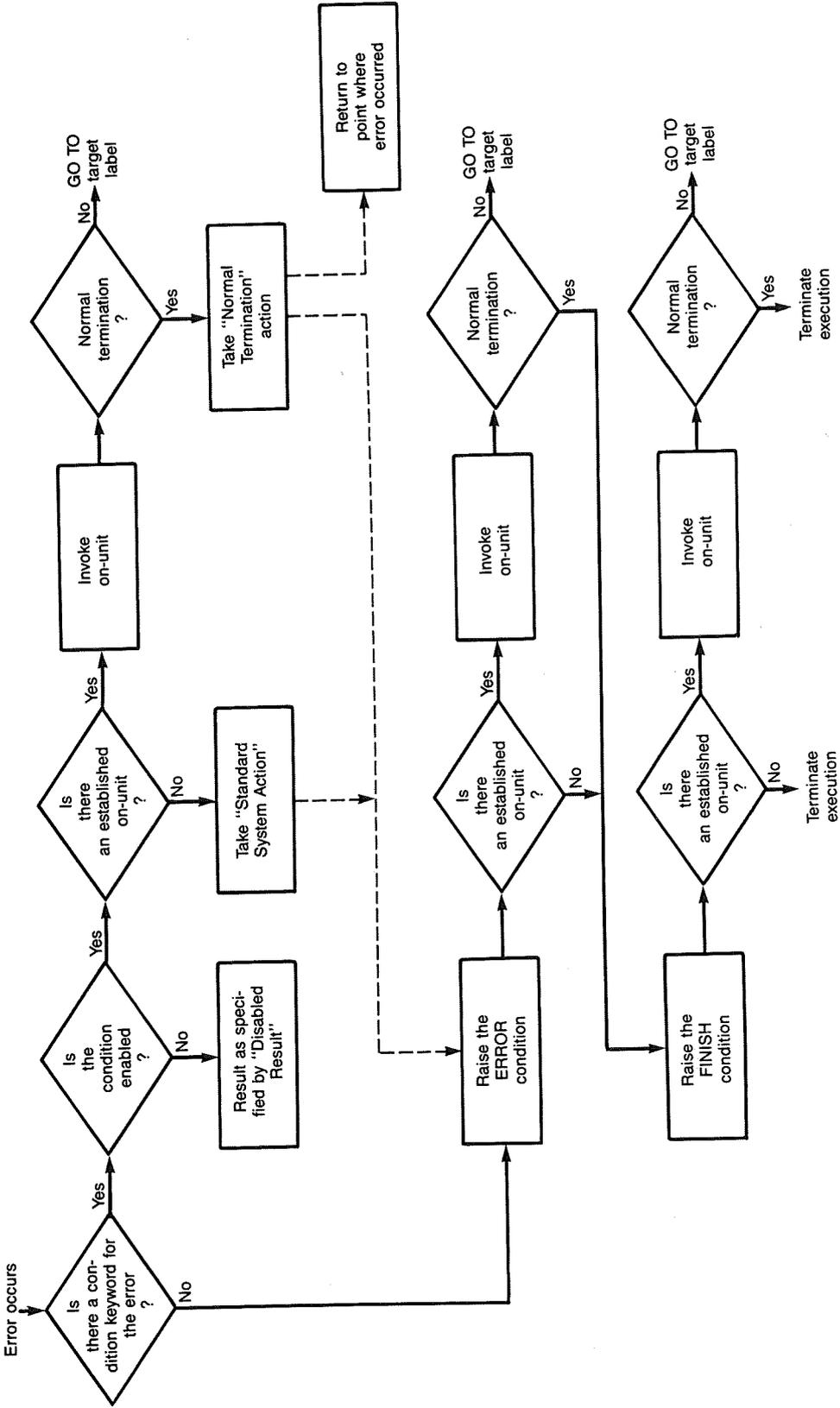


Figure 13-3
Condition Handling

The AREA Condition

Description: PL/I raises the AREA condition in either of the following cases:

- Your program executes an ALLOCATE statement with the IN option, but the allocation fails because the specified area does not have enough space remaining for the allocation. This can occur if you use the ALLOCATE statement to specify a storage area that is greater than one segment.
- Your program executes an assignment of one area to another, but the target area is too small to hold all the allocated space in the source area.

Enablement Status: The AREA condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the AREA condition, PL/I raises the ERROR condition.

Normal Termination Action: If an AREA on-unit terminates normally, PL/I takes the following action:

- If the on-unit was invoked as the result of an error in an ALLOCATE statement, PL/I returns to the ALLOCATE statement, re-evaluates the IN option, and reattempts the allocation. Therefore, in this case you should not terminate the on-unit normally unless your on-unit takes steps to make the area specified by the IN option larger. Normal termination without enlargement of the area results in an infinite loop.
- If the on-unit was invoked as the result of an error in an area assignment, PL/I returns to the point of interrupt and continues the assignment, but with an undefined result.

The CONDITION Condition

Syntax: CONDITION(reference)

Abbreviation: COND for CONDITION

Description: This is the user-defined condition, and is described earlier in the section on the SIGNAL statement. This condition cannot be raised by a program error, but only by the SIGNAL statement.

The CONVERSION Condition

Abbreviation: CONV for CONVERSION

Description: The CONVERSION condition is raised whenever a conversion from CHARACTER or pictured-character fails because the source string contains an invalid character. Examples of the CONVERSION condition are given earlier in this chapter.

Enablement Status: The CONVERSION condition is enabled by default. You may disable it by means of a condition prefix specifying NOCONVERSION.

Disabled Result: If a conversion error occurs at a point in your program where the CONVERSION condition is disabled, PL/I simply misinterprets the invalid character, and continues with the conversion. The result of the conversion is unpredictable.

Standard System Action: If there is no established on-unit for the CONVERSION condition, PL/I raises the ERROR condition.

Normal Termination Action: In case of normal termination of a CONVERSION on-unit invoked as the result of a conversion error, PL/I returns to the statement in which the conversion error occurred, and reattempts the conversion from the beginning of the source string. In order for this to work properly, the CONVERSION on-unit should have used the ONCHAR and ONSOURCE pseudovariables to modify the conversion source string so that the new conversion attempt will succeed. As examples given earlier in this chapter illustrate, it is very easy for normal termination of a CONVERSION on-unit to result in an infinite loop. For this reason, it is recommended that all CONVERSION on-units terminate abnormally (with a GO TO out of the on-unit).

The ENDFILE Condition

Syntax: ENDFILE(file), where file is an identifier or expression with the FILE attribute.

Description: PL/I raises the ENDFILE condition when a GET or READ statement fails because input has reached end of file. In the case of GET statement input, if end of file occurs in the middle of a stream data item, then ERROR is raised instead; ENDFILE is raised only if the end of file occurs between data items.

PL/I Reference Guide

See Chapters 11 and 12 for more information on the ENDFILE condition.

Enablement Status: The ENDFILE condition is enabled by default and you may not disable it.

Standard System Action: If there is no on-unit established for the ENDFILE condition, PL/I raises the ERROR condition.

Normal Termination Action: If an ENDFILE on-unit invoked as the result of an end-of-file error terminates normally, PL/I continues execution of the program with the statement following the GET or READ statement that failed because of the end-of-file error. The end-of-file status remains set so that subsequent READS of the same file with no intervening CLOSE also cause the condition to occur.

The ENDPAGE Condition

Syntax: ENDPAGE(file), where file is an identifier or expression with the FILE attribute.

Description: Although, strictly speaking, ENDPAGE is not an ERROR condition, PL/I handles the ENDPAGE condition just like other conditions that do arise from errors.

PL/I raises the ENDPAGE condition as the result of an operation in a PUT statement to a PRINT file that attempts to begin a new line of output beyond the page size for the file. See Chapter 11 for examples.

Enablement Status: The ENDPAGE condition is enabled by default and you may not disable it.

Standard System Action: If there is no on-unit established for the ENDPAGE condition, PL/I skips to a new page and then returns to the PUT statement to continue output.

Normal Termination Action: If an ENDPAGE on-unit terminates normally, PL/I returns to the PUT statement to continue output. See Chapter 11 for examples using an ENDPAGE on-unit with normal return to print page headings.

The ERROR Condition

Description: PL/I raises the ERROR condition under the conditions described at the beginning of this section, usually because of an error for which there is no condition keyword, or as the standard system action or normal return action for another condition. Just prior to invocation of the condition, the value of ONCODE is set to a code indicating which error occurred. See Appendix F for a list of these codes.

ON ERROR overrides any previously established on-units. If other on-units are used in the same block, ON ERROR should be processed first.

Enablement Status: The ERROR condition is enabled by default and you may not disable it.

Standard System Action: If there is no on-unit established for the ERROR condition, PL/I raises the FINISH condition.

Normal Termination Action: If an ERROR on-unit invoked as the result of a program error terminates normally, PL/I raises the FINISH condition.

The FINISH Condition

Description: PL/I raises the FINISH condition as the result of any statement or error situation that might terminate your program's execution. These cases are as follows:

- If your program executes a STOP statement.
- If your program executes a RETURN statement from the main procedure of your program.
- If your program executes the END statement of the main procedure of your program.
- Raising the FINISH condition is the standard system action and normal termination action for the ERROR condition.

Enablement Status: The FINISH condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the FINISH condition, PL/I simply terminates your program.

Normal Termination Action: If a FINISH on-unit terminates normally, PL/I terminates execution of your program.

If your on-unit terminates abnormally, execution of your program continues from the point that the GO TO statement specifies. Therefore, it is possible that your program will enter an infinite loop and will never terminate.

The FIXEDOVERFLOW Condition

Note

In Prime implementations of PL/I, you must specify the -OVERFLOW compiler option to enable hardware detection of the FIXEDOVERFLOW condition.

Abbreviation: FOFL for FIXEDOVERFLOW

Description: PL/I raises the FIXEDOVERFLOW condition when the hardware detects that a fixed-point binary or decimal result is too large to fit into the hardware register or the decimal field. The hardware register limit is implementation-defined; the decimal field limit is user-defined. The maximum number of digits is 31 for FIXED BINARY and 14 for FIXED DECIMAL.

Enablement Status: The FIXEDOVERFLOW condition is enabled by default. You may disable it by means of a condition prefix specifying NOFIXEDOVERFLOW.

Disabled Result: If a fixed overflow error occurs at a point in your program where the FIXEDOVERFLOW condition is disabled, PL/I simply ignores the error, and the result of the computation is undefined.

Standard System Action: If there is no established on-unit for the FIXEDOVERFLOW condition, PL/I raises the ERROR condition.

Normal Termination Action: If a FIXEDOVERFLOW on-unit invoked as the result of a fixed overflow error terminates normally, PL/I raises the ERROR condition.

The KEY Condition

Syntax: KEY(file), where file is an identifier or expression with the FILE attribute.

Description: PL/I raises the KEY condition on operations on KEYED files, when the value specified in the KEY or KEYFROM option is illegal or causes any sort of error. Typical error situations are as follows:

- An attempt was made to add a record with a duplicate key to a file.
- A record with the specified key cannot be found.
- The CHARACTER string value specified with the KEY or KEYFROM option has an invalid format.

Enablement Status: The KEY condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the KEY condition, PL/I raises the ERROR condition.

Normal Termination Action: If a KEY on-unit invoked as the result of an error in an input/output statement terminates normally, PL/I continues execution with the statement following the input/output statement that contains the key error.

The NAME Condition

Syntax: NAME(file), where file is an identifier or expression with the FILE attribute.

Description: PL/I raises the NAME condition when a GET DATA statement fails because of an invalid variable reference in the input stream. The variable reference can be invalid because it contains illegal characters, because the variable name is not recognized as legal for the statement, or because a subscript list is missing or invalid.

Enablement Status: The NAME condition is enabled by default and you may not disable it.

PL/I Reference Guide

Standard System Action: If there is no on-unit established for the NAME condition, PL/I continues execution of the GET DATA statement by continuing with the next assignment in the input stream.

Normal Termination Action: If a NAME on-unit invoked as the result of an error in a GET DATA statement terminates normally, PL/I continues execution of the GET DATA statement with the next assignment in the input stream.

The OVERFLOW Condition

Abbreviation: OFL for OVERFLOW

Description: PL/I raises the OVERFLOW condition during evaluation of an expression when an intermediate FLOAT computation gives rise to a value whose characteristic (exponent) is positive and too large to be supported for a FLOAT value on a Prime computer.

Enablement Status: The OVERFLOW condition is enabled by default and may be disabled by means of a condition prefix specifying NOOVERFLOW.

Disabled Result: If an overflow error occurs at a point in your program where the OVERFLOW condition is enabled, PL/I simply ignores the error, and the result of the computation is undefined.

Standard System Action: If there is no established on-unit for the OVERFLOW condition, PL/I raises the ERROR condition.

Normal Termination Action: If an OVERFLOW on-unit invoked as the result of a floating-point overflow error terminates normally, PL/I raises the ERROR condition.

The RECORD Condition

Syntax: RECORD(file), where file is an identifier or expression with the FILE attribute.

Description: PL/I raises the RECORD condition whenever a READ, WRITE, REWRITE, or LOCATE statement fails because the size of the record being transmitted to the external file or device, as determined by the file specifications, is inconsistent with the size of the FROM or INTO variable. Typical errors are as follows:

- In a READ statement, the size of the record being transmitted does not equal the size of the INTO variable.
- In a WRITE, REWRITE, or LOCATE statement, the size of the record being transmitted is unacceptable for the specified file.

Enablement Status: The RECORD condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the RECORD condition, PL/I raises the ERROR condition.

Normal Termination Action: If a RECORD on-unit invoked as the result of an error in a READ, WRITE, REWRITE, or LOCATE statement terminates normally, PL/I continues execution with the statement following the one that caused the error.

The SIZE Condition

Description: PL/I raises the SIZE condition during arithmetic conversion, when a value is too large to fit into the target data type.

To understand the difference between the SIZE condition and the FIXEDOVERFLOW condition, consider the following program:

```
(SIZE): COND: PROC OPTIONS (MAIN);
DECLARE (A, B) FIXED DECIMAL(5);
DECLARE C FIXED BINARY(31);
A = 99999;
B = A * A * A;           /* RAISES FIXEDOVERFLOW */
C = A + 1;
B = C;                   /* RAISES SIZE */
END COND;
```

The fifth statement of this program raises the `FIXEDOVERFLOW` condition during the computation of $A * A * A$, because the result of this computation exceeds 14 decimal digits, which is the maximum number of digits that Prime PL/I supports for `FIXED DECIMAL`. On the other hand, the seventh statement of the program does not raise `FIXEDOVERFLOW`, because the value of $A + 1$ is 100000, which is small enough to be handled by the Prime PL/I `FIXED BINARY` data type. However, in attempting to convert this value to the variable `B`, PL/I raises the `SIZE` condition (if enabled), because `B` cannot accommodate more than five decimal digits.

Enablement Status: `SIZE` is disabled by default, but you may enable it by means of a condition prefix specifying the `SIZE` condition. For more information, see the section Enabling Conditions for Debugging earlier in this chapter.

Disabled Result: If a size error occurs with the `SIZE` condition disabled, PL/I ignores the error and makes an invalid assignment, with the result that the `FIXED` target variable has an unpredictable value.

Standard System Action: If there is no established on-unit for the `SIZE` condition, PL/I raises the `ERROR` condition.

Normal Termination Action: If a `SIZE` on-unit invoked as the result of a size error terminates normally, PL/I raises the `ERROR` condition.

The STORAGE Condition

Description: PL/I raises the `STORAGE` condition when a storage allocation fails because there is insufficient storage available for your program.

Enablement Status: The `STORAGE` condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the `STORAGE` condition, PL/I raises the `ERROR` condition.

Normal Termination Action: If a `STORAGE` on-unit invoked as the result of an allocation failure terminates normally, PL/I returns to the point where the allocation failed and reattempts the allocation. For this reason, before a `STORAGE` on-unit terminates normally it should free some storage so that the allocation reattempt will succeed; otherwise, there may be an infinite loop.

The STRINGRANGE Condition

Abbreviation: STRG for STRINGRANGE

Description: PL/I raises the STRINGRANGE condition whenever a reference to the SUBSTR built-in function or pseudovisible fails because the second argument or the third argument is out of range. Specifically, a reference to

SUBSTR(c, i, j)

either as a built-in function or a pseudovisible, raises STRINGRANGE, unless it is true that

$$0 \leq i - 1 \leq j + i - 1 \leq \text{LENGTH}(c)$$

A reference to

SUBSTR(c, i)

raises the STRINGRANGE condition unless

$$0 \leq i - 1 \leq \text{LENGTH}(c)$$

Enablement Status: The STRINGRANGE condition is disabled by default. You may enable it by means of a condition prefix specifying STRINGRANGE. For more information, see the section Enabling Conditions for Debugging earlier in this chapter.

Disabled Result: If the second or third argument to SUBSTR is out of range, and the STRINGRANGE condition is disabled, PL/I ignores the error and either fetches or assigns the string value as if the argument to SUBSTR were correct. This means that PL/I fetches characters from or stores characters in storage areas that are not part of the storage area associated with the first argument to SUBSTR. In this case the results are unpredictable. In the case of the SUBSTR pseudovisible, assignment may destroy crucial constants or pointers, so that your entire program may execute unpredictably after that. For this reason, it is recommended that you enable the STRINGRANGE condition during the debugging phases of your program development, as described in the section Enabling Conditions for Debugging earlier in this chapter.

Standard System Action: If there is no STRINGRANGE on-unit, PL/I raises the ERROR condition.

Normal Termination Action: If a STRINGRANGE on-unit invoked as the result of an error in the SUBSTR built-in function or pseudovisible terminates normally, PL/I raises the ERROR condition.

The STRINGSIZE Condition

Abbreviation: STRZ for STRINGSIZE

Description: PL/I raises the STRINGSIZE condition whenever either of the following occurs:

- In a PUT EDIT statement, the data format item is one of B(w), Bl(w), B2(w), B3(w), or B4(w), and the corresponding scalar data item, after being converted to BIT, contains too many bits to be printed in the specified width w.
- In an assignment to a CHARACTER, pictured-character, or BIT variable, the source string is longer than the maximum length of the assignment target.

Enablement Status: STRINGSIZE is enabled by default. You may disable it by means of a condition prefix specifying NOSTRINGSIZE, or by specifying the -NO_STRINGSIZE compiler option.

Disabled Result: PL/I detects a STRINGSIZE situation even when the STRINGSIZE situation is disabled, and takes the standard system action, but does not invoke a STRINGSIZE on-unit even if one has been established.

Standard System Action: If either of the STRINGSIZE errors described above occurs, and if the STRINGSIZE condition is disabled, or if there is no established on-unit for the STRINGSIZE condition, or if a STRINGSIZE on-unit terminates normally, then PL/I truncates the source string to the appropriate length for either the BIT data item of PUT EDIT or the assignment target.

Normal Termination Action: If a STRINGSIZE on-unit raised as the result of a STRINGSIZE error terminates normally, PL/I truncates the source string as described above under Standard System Action.

The SUBSCRIPTRANGE Condition

Abbreviation: SUBRG for SUBSCRIPTRANGE

Description: PL/I raises the SUBSCRIPTRANGE condition whenever a subscript computation is made and it is found that the computed subscript value is outside the bounds permitted for that subscript position in that array variable.

Enablement Status: The SUBSCRIPTRANGE condition is disabled by default. You may enable it by means of a condition prefix specifying the SUBSCRIPTRANGE condition. For more information, see the section Enabling Conditions for Debugging earlier in this chapter.

Disabled Result: If a SUBSCRIPTRANGE error occurs at a point in your program where the SUBSCRIPTRANGE condition is disabled, PL/I ignores the SUBSCRIPTRANGE error, and either fetches or stores the data value as if the subscript value were correct. This means that PL/I fetches data from or stores data in storage areas that are not part of the storage area associated with the array variable. This means that the results are unpredictable. In the case of assignment to an array variable with an out-of-bounds subscript value, the assignment operation may destroy crucial constants or pointers, so that your entire program may execute unpredictably after that. For this reason, we recommend that you enable the SUBSCRIPTRANGE condition during the debugging phases of your program development, as described in the section Enabling Conditions for Debugging earlier in this chapter.

Standard System Action: If there is no established on-unit for the SUBSCRIPTRANGE condition, PL/I raises the ERROR condition.

Normal Termination Action: If a SUBSCRIPTRANGE on-unit raised as the result of a SUBSCRIPTRANGE error terminates normally, PL/I invokes the ERROR condition.

The TRANSMIT Condition

Syntax: TRANSMIT(file), where file is an identifier or expression with the FILE attribute.

Description: PL/I raises the TRANSMIT condition whenever an input/output statement fails because of a hardware data transmission error.

Enablement Status: The TRANSMIT condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the TRANSMIT condition, PL/I raises the ERROR condition.

Normal Termination Action: If a TRANSMIT on-unit invoked as the result of a hardware data transmission error in an input/output statement terminates normally, PL/I returns to the point where the input/output operation failed and continues execution. It is quite likely that the next input/output operation will raise the TRANSMIT condition again.

The UNDEFINEDFILE Condition

Syntax: UNDEFINEDFILE(file), where file is an identifier or an expression with the FILE attribute.

Abbreviation: UNDF for UNDEFINEDFILE

Description: PL/I raises the UNDEFINEDFILE condition whenever a file opening, whether explicit (with an OPEN statement) or implicit, fails for any reason. Possible reasons include the following:

- An INPUT or UPDATE file is not found.
- The attributes specified in the DECLARE statement for a file conflict with the attributes specified in the OPEN statement, resulting in an open attribute merge inconsistency.
- The TITLE option string in the OPEN statement has an invalid format.

Enablement Status: The UNDEFINEDFILE condition is enabled by default and you may not disable it.

Standard System Action: If there is no established on-unit for the UNDEFINEDFILE condition, PL/I raises the ERROR condition.

Normal Termination Action: If an UNDEFINEDFILE on-unit invoked as the result of a failure in an explicit OPEN statement terminates normally, PL/I returns to the point where the error occurred and continues execution. The file is left open only if the on-unit successfully opened the file; otherwise, the file is left closed.

PL/I CONDITION HANDLING

If an UNDEFINEDFILE on-unit invoked because of an implicit file opening failure in a GET, PUT, READ, WRITE, REWRITE, DELETE, or LOCATE statement terminates normally, PL/I takes action as follows:

- If the on-unit successfully opened the file, PL/I returns to the point where the error was detected and continues execution of the statement.
- If the file is still closed after normal termination of the on-unit, PL/I raises the ERROR condition.

The UNDERFLOW Condition

Abbreviation: UFL for UNDERFLOW

Description: PL/I raises the UNDERFLOW condition during evaluation of an expression, when an intermediate FLOAT computation gives rise to a value whose characteristic (exponent) is negative and too large in absolute value for a FLOAT value on Prime computers.

Enablement Status: The UNDERFLOW condition is enabled by default. You may disable it by means of a condition prefix specifying NOUNDERFLOW.

Disabled Result: If a floating-point UNDERFLOW condition occurs at a point in your program where the UNDERFLOW condition is disabled, PL/I ignores the UNDERFLOW error and sets the value of the invalid FLOAT computation to 0.

Standard System Action: If there is no established on-unit for the UNDERFLOW condition, PL/I returns to the point where the error is detected to complete the computation, after setting the value of the invalid FLOAT computation to 0.

Normal Termination Action: If an UNDERFLOW on-unit invoked as the result of a floating-point UNDERFLOW error terminates normally, PL/I returns to the point where the error was detected to complete the computation, after setting the value of the invalid FLOAT computation to 0.

The ZERODIVIDE Condition

Abbreviation: ZDIV for ZERODIVIDE

Description: PL/I raises the ZERODIVIDE condition during evaluation of an expression when division by 0 is attempted.

Enablement Status: The ZERODIVIDE condition is enabled by default. You may disable it by means of a condition prefix specifying NOZERODIVIDE.

Disabled Result: If your program attempts division by 0 at a point where the ZERODIVIDE condition is disabled, PL/I simply ignores the division error, and the result of the computation is undefined.

Standard System Action: If there is no established on-unit for the ZERODIVIDE condition, PL/I raises the ERROR condition.

Normal Termination Action: If a ZERODIVIDE on-unit invoked as the result of an attempt to divide by 0 in a computation terminates normally, PL/I returns to the point in the computation where the error was detected and continues execution from that point. The result of the invalid attempt to divide by 0 is unpredictable.

BUILT-IN FUNCTIONS RELATED TO ON-UNITS

PL/I may raise a given condition under many different circumstances, and for many different errors. This means that if you write an on-unit for a given condition, the on-unit must be capable of recovering from many different types of errors. PL/I provides several built-in functions that you may use within an on-unit to determine what problem caused the on-unit to be invoked.

One of the most useful of these is ONCODE(). This built-in function returns an integer value equal to the internal code for the error that invoked the on-unit. The ONCODE built-in function can be used in an on-unit for any condition, to determine the specific error that caused the on-unit invocation. A list of error codes returned by ONCODE is in Appendix F.

The built-in functions for condition handling are shown in Table 13-1. Each of these built-in functions returns a character value containing the information specified in the table.

Each of these built-in functions should be used only within on-units for the conditions shown in the table. If you use one of these built-in functions elsewhere than within an on-unit for the condition or conditions specified for that built-in function in the table, or if you use the built-in function within an on-unit invoked as the result of a SIGNAL statement, then PL/I returns the null string as the value for that built-in function invocation. The following are two exceptions to this rule:

- ONCHAR() returns a CHARACTER(1) string containing a blank character, rather than a null string.
- If ONLOC() is used in any context, it returns the value indicated by Table 13-1.

ONCHAR and ONSOURCE are also pseudovariables, as described earlier in the section Examples of ON CONVERSION.

Table 13-1
Built-in Functions for Condition Handling

Built-in Function	To Be Used in On-units for These Conditions	CHARACTER String Value Returned by Built-in Function
ONCHAR ()	CONVERSION	Character that caused conversion error
ONFIELD ()	NAME	Invalid input stream chars to GET DATA
ONFILE ()	ENDFILE, ENDPAGE, KEY, NAME, RECORD, TRANSMIT, UNDEFINEDFILE, CONVERSION	Name of FILE constant on which error occurred
ONKEY ()	TRANSMIT, RECORD, KEY	Invalid key value, or key of record causing error
ONLOC ()	Any	Name of entry point in PROCEDURE in which condition was raised
ONSOURCE ()	CONVERSION	Entire source string for which the conversion error occurred
ONCODE ()	Any	See Appendix F

14

Built-in Functions and Pseudovariabes

This chapter discusses the use of built-in functions and pseudovariabes. It also contains a complete list of all built-in functions and pseudovariabes supported by Prime PL/I.

You may use a built-in function reference in any expression in any statement of your program. The value returned by a built-in function may have any data type, as determined by the rules for that built-in function. It is also possible for a built-in function to return a nonscalar aggregate value (such as an array value or a structure value), as this chapter explains.

Certain built-in functions may also be used in a context where a value is assigned to the function. When used in this way, such a built-in function is called a pseudovariabes. The built-in functions that are also pseudovariabes are as follows: IMAG, ONCHAR, ONSOURCE, PAGENO, REAL, SUBSTR, and UNSPEC. The use of pseudovariabes is described at the end of this chapter.

ARGUMENTS TO BUILT-IN FUNCTIONS

In this section, some general rules for the arguments of built-in functions are described. In the complete list of built-in functions given later in this chapter, the specific rules for arguments of each built-in function are described in detail.

Built-in Functions With No Arguments

Several built-in functions take no arguments at all. You must take some special precautions when you use these built-in functions.

Consider the following statement:

```
PUT LIST( DATE );
```

The writer of this statement intended to invoke the DATE built-in function, which takes no arguments and prints out the current date. Unfortunately, in this statement, PL/I will interpret the identifier DATE as an ordinary variable, rather than as the name of the built-in function DATE.

If you wish PL/I to interpret DATE as a built-in function, the identifier DATE must be given the BUILTIN attribute. There are two different ways to do this. One is to write an empty pair of parentheses after the identifier DATE, as in the following statement:

```
PUT LIST( DATE ( ) );
```

The appearance of a left parenthesis following the identifier DATE causes PL/I to declare DATE contextually to have the BUILTIN attribute.

The second method is to use the DECLARE statement to give the function identifier the BUILTIN attribute. For example, the statement

```
DECLARE DATE BUILTIN;
```

gives DATE the BUILTIN attribute and permits you to reference the built-in function DATE without using the empty pair of parentheses.

You must take similar precautions with each of the built-in functions that take no arguments. These are as follows: COLLATE, DATE, EMPTY, NULL, ONCHAR, ONCODE, ONFIELD, ONFILE, ONKEY, ONLOC, ONSOURCE, and TIME.

Aggregate Arguments to Built-in Functions

Usually an argument to a built-in function is a scalar variable or scalar expression. However, most built-in functions permit you to use nonscalar arguments, such as array or structure expressions. Different built-in functions have different rules for handling arguments whose aggregate types are nonscalar. These rules can be classified as follows:

- Most built-in functions handle aggregate arguments in the same way that arithmetic and string operators handle aggregates, as described in Chapter 6. This is discussed more fully in the next section, The General Rule for Aggregates.
- The array-handling built-in functions are a group of built-in functions, at least one argument of which must be an array. Each of these functions returns a scalar result. They are DIMENSION, HBOUND, and LBOUND, the first argument of which must be an array, and DOT, PROD, and SUM, the first two arguments of which must be arrays.
- ADDR, ALLOCATION, OFFSET, and POINTER are storage-handling built-in functions that use only allocation information about the argument, and so are indifferent to the aggregate type of the argument.
- STRING, SOME, and EVERY are built-in functions that permit a string aggregate as an argument and that treat the aggregate as one long string.
- HIGH, LOW, UNSPEC, LINENO, PAGENO are built-in functions that forbid arguments whose aggregate type is nonscalar.

Some built-in functions combine the rules above, following different rules for different arguments. For example, the COPY built-in function permits aggregates in the first argument, following the first rule above, but forbids nonscalar aggregates in the second argument (last rule).

The General Rule for Aggregates

As stated in the first rule above, most built-in functions handle nonscalar aggregate arguments in the same way that arithmetic operators handle them, as described in Chapter 6. In the complete list of built-in functions given later in this chapter, such functions will be described as following the general rule for aggregates. This section explains more precisely what this means.

First, consider the built-in function ABS. If the argument of ABS is aggregate, ABS returns a value of the same aggregate type. For example, if S is an array of structures, ABS(S) returns an array of structures, each of whose elements is computed by taking the absolute value of the corresponding element of S.

A built-in function like MAX is more complicated, since it can have several aggregate arguments. In this case, the rules for combining different aggregate types, as described in Chapter 6, apply. For example, if T is a simple array, and S is an array of structures, then PL/I computes MAX(T, S) as follows:

1. PL/I checks to see whether T and S have compatible aggregate types. In this particular case, PL/I must check to see that T and S have identical array bounds.
2. PL/I promotes each argument to the common derived aggregate type of all the arguments. In this case, PL/I must promote T to an array of structures.
3. As the value of the built-in function, PL/I returns an aggregate value whose aggregate type is the common derived aggregate type of the arguments. In this case, PL/I returns an array of structures, each of whose elements is computed by applying the MAX built-in function to the corresponding elements of S and promoted value of T.

In the built-in function list, when the general rule for aggregates is referred to for certain arguments, the rules outlined in Chapter 6 are meant, and are illustrated above.

The Derived Data Types and Converted Precision

The rules for many of the built-in functions require that the arguments have numeric data types. If you reference such a function with a nonnumeric argument, PL/I must convert the argument to a numeric data type before computation of the function can begin.

For example, suppose C is a CHARACTER variable, and you reference ABS(C). Then PL/I evaluates this reference as follows:

1. PL/I converts C to the derived base, scale, and mode of the data type of C, and the converted precision of the data type of C. These terms are defined in Chapter 6. In this particular case, PL/I converts C to a new value, X, whose data type is FIXED DECIMAL(14, 0) REAL.
2. PL/I then computes the absolute value of X.

Other built-in functions require that the arguments be strings, and still others require not only that the arguments all be numeric or all be string, but also that they all have the same numeric or string data type. In the complete list of built-in functions given later in this chapter, when the data type conversions of the arguments are described, the following terms are used. These terms are defined in Chapter 6.

- If it is necessary to convert a nonnumeric data type to a numeric data type, the numeric data type usually has the derived base, scale, and mode of the original data type.
- If it is necessary to convert two or more arguments to the same numeric data type, that common data type usually has the common derived base, scale, and mode of the data types of the arguments.
- Once you know that you must convert a data type to a specific base, scale, and mode, you need to know what the precision of the numeric data type must be. This is usually the converted precision of the data type.
- If a nonstring value must be converted to a string data type (CHARACTER or BIT), the string type is usually the derived string type of the data type.
- If two or more arguments of a built-in function must be converted to the same string data type, the target data type is usually the common derived string type of the arguments.

Arguments That Specify Precision: p and q

Several of the built-in functions have arguments to specify the arithmetic precision of the result to be returned by the built-in function. For example, if X and Y are FIXED DECIMAL scalar variables, a reference to

```
ADD(X, Y, 7, 2)
```

returns the value of X + Y, with a target data type of FIXED DECIMAL(7, 2).

In the complete list of built-in functions given later in this chapter, the letters p and q are used to indicate these precision arguments. As is indicated in the description of each of these built-in functions, q is always an optional argument, and sometimes both p and q are optional.

For these built-in functions, the following rules apply:

- The arguments p and q, if specified, may not be arbitrary PL/I expressions, but must be decimal integer constants. The value of p must be positive, while q may be positive, zero, or negative.
- The base, scale, and mode of the value returned by the function do not depend upon the values of p and q, or on whether you specify p or q. Rather, the base, scale, and mode of the result are computed as the derived common base, scale, and mode of the argument or arguments that precede the argument p in the argument list.
- If the derived common scale is FLOAT, it is illegal to specify the argument q. If you specify the argument p when the derived common scale is FLOAT, the precision of the value returned by the built-in function is (p).
- If the derived common scale is FIXED, the argument q is optional. The default scale factor is 0. If you specify both p and q, the precision of the value returned by the built-in function is (p, q); if you specify only p, the precision of the value returned by the function is (p, 0).
- A consequence of the above rules is that the value of p may not exceed the maximum number of digits permitted for the derived common base and scale for the argument or arguments that precede p in the argument list. The maximum values for p are 31 for FIXED BINARY, 14 for FIXED DECIMAL, 47 for FLOAT BINARY, and 14 for FLOAT DECIMAL.
- If you specify neither p nor q (in those built-in functions where p is an optional argument), the precision of the result returned by the function is the converted common precision of the argument or arguments.

In the descriptions of these built-in functions, the current section is referred to whenever p and q are arguments.

CLASSIFICATION AND SUMMARY OF BUILT-IN FUNCTIONS

This section classifies the functions into groups with similar functionality. In the following lists, arguments that are enclosed in square brackets are optional.

The Arithmetic Built-in Functions

These functions perform simple arithmetic manipulations on the arguments. (Note that some of these built-in functions are considered to be mathematical built-in functions when the arguments are COMPLEX rather than REAL. ABS is an example of such a function.)

<u>Function</u>	<u>Returns</u>
ABS(x)	The absolute value of <u>x</u>
ADD(x, y, p[, q])	Value of $x + y$
BINARY(x[, p[, q]])	Result of converting <u>x</u> to base BINARY
CEIL(x)	Smallest integer greater than or equal to <u>x</u>
DECIMAL(x[, p[, q]])	Result of converting <u>x</u> to base DECIMAL
DIVIDE(x, y, p[, q])	Value of x/y
FIXED(x, p[, q])	Result of converting <u>x</u> to scale FIXED
FLOAT(x, p)	Result of converting <u>x</u> to scale FLOAT
FLOOR(x)	Largest integer less than or equal to <u>x</u>
MAX(x1, x2, ..., xn)	Maximum of the arguments
MIN(x1, x2, ..., xn)	Minimum of the arguments
MOD(x, y)	Remainder when <u>x</u> is divided by <u>y</u>
MULTIPLY(x, y, p[, q])	Value of $x * y$
PRECISION(x, p[, q])	Result of converting <u>x</u> to specified precision
ROUND(x, n)	Result of rounding FIXED argument to specified digit position; or FLOAT argument to specified number of significant digits
SIGN(x)	+1, 0, -1, according to whether <u>x</u> is positive, zero, or negative

<u>Function</u>	<u>Returns</u>
SUBTRACT(x, y, p[, q])	Value of $x - y$
TRUNC(x)	Integer obtained by truncating <u>x</u>

The Mathematical Built-in Functions

In most cases, PL/I computes the value of a mathematical built-in function reference by means of a polynomial approximation.

<u>Function</u>	<u>Returns</u>
ACOS(x)	The arc cosine of <u>x</u> , with the result measured in radians
ASIN(x)	The arc sine of <u>x</u> , with the result measured in radians
ATAN(x)	The arc tangent of <u>x</u> , with the result measured in radians
ATAN(x, y)	The arc tangent of y/x , with the result measured in radians
ATAND(x)	The arc tangent of <u>x</u> , with the result measured in degrees
ATAND(x, y)	The arc tangent of y/x , with the result measured in degrees
ATANH(x)	The arc hyperbolic tangent of <u>x</u>
COMPLEX(x, y)	COMPLEX value whose real part is <u>x</u> and whose imaginary part is <u>y</u>
CONJG(x)	The COMPLEX conjugate of <u>x</u>
COS(x)	The cosine of <u>x</u> , with <u>x</u> measured in radians
COSD(x)	The cosine of <u>x</u> , with <u>x</u> measured in degrees
COSH(x)	The hyperbolic cosine of <u>x</u>
ERF(x)	The error function of <u>x</u>
ERFC(x)	The complement error function of <u>x</u> , $1 - \text{ERF}(x)$

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

<u>Function</u>	<u>Returns</u>
EXP(x)	e^x
IMAG(x)	The imaginary part of x
LOG(x)	The natural logarithm (logarithm to base e) of x
LOG10(x)	The common logarithm (logarithm to base 10) of x
LOG2(x)	The logarithm to base 2 of x
REAL(x)	The real part of x
SIN(x)	The sine of x , with x measured in radians
SIND(x)	The sine of x , with x measured in degrees
SINH(x)	The hyperbolic sine of x
SQRT(x)	The square root of x
TAN(x)	The tangent of x , with x measured in radians
TAND(x)	The tangent of x , with x measured in degrees
TANH(x)	The hyperbolic tangent of x

The String-handling Built-in Functions

These built-in functions either take string arguments or return string values.

<u>Function</u>	<u>Returns</u>
AFTER(s, c)	The portion of s after substring c
BEFORE(s, c)	The portion of s before substring c
BIT(x[, n])	The result of converting x to a BIT string of length n

PL/I Reference Guide

<u>Function</u>	<u>Returns</u>
BOOL(x, y, c)	The result of performing logical operations specified by <u>c</u> on BIT strings <u>x</u> and <u>y</u>
BYTE(x)	(Prime extension) CHARACTER(1) equivalent of numeric value <u>x</u>
CHARACTER(x[, n])	Result of converting <u>x</u> to a CHARACTER string of length <u>n</u>
COLLATE()	String containing the entire collating sequence
COPY(s, n)	String containing <u>n</u> concatenated copies of string <u>s</u>
DECAT(s, c, t)	Result of concatenating parts of <u>s</u> and <u>c</u> specified by bit string <u>t</u>
EVERY(x)	Logical AND of all bits in <u>x</u>
HIGH(n)	String of length <u>n</u> containing <u>n</u> occurrences of the highest character in the collating sequence
INDEX(s, c)	Position of substring <u>c</u> in <u>s</u>
LENGTH(s)	Length of string <u>s</u>
LOW(n)	String of length <u>n</u> containing <u>n</u> occurrences of the lowest character in the collating sequence
RANK(x)	(Prime extension) Numeric equivalent of CHARACTER(1) value of <u>x</u>
REVERSE(s)	String obtained by reversing the order of all characters or bits in <u>s</u>
SOME(x)	Logical OR of all bits in <u>x</u>
STRING(x)	Result of concatenating all elements of aggregate <u>x</u>
SUBSTR(s, m, n)	Substring of <u>s</u> starting at position <u>m</u> and going for <u>n</u> characters or bits

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

<u>Function</u>	<u>Returns</u>
SUBSTR(<u>s</u> , <u>m</u>)	Substring of <u>s</u> starting at position <u>m</u> and going to the end of <u>s</u>
TRANSLATE(<u>s</u> , <u>r</u> , <u>t</u>)	Result of translating characters in <u>s</u> that are in the set <u>t</u> to the corresponding character in <u>r</u>
TRIM(<u>s</u> , <u>b</u> [, <u>t</u>])	Result of removing blanks or occurrences of <u>t</u> from string <u>s</u> , in pattern indicated by <u>b</u>
VERIFY(<u>s</u> , <u>t</u>)	Position of the first character in <u>s</u> not also in string <u>t</u>

The Array-handling Built-in Functions

Each of these built-in functions has one or two arguments that must be an array. Each function returns a scalar value.

<u>Function</u>	<u>Returns</u>
DIMENSION(<u>x</u> , <u>n</u>)	The dimension size of array <u>x</u> in the <u>n</u> th dimension
DOT(<u>x</u> , <u>y</u> [, <u>p</u> [, <u>q</u>]])	Dot product of the arrays <u>x</u> and <u>y</u>
HBOUND(<u>x</u> , <u>n</u>)	Upper bound of array <u>x</u> in the <u>n</u> th dimension
LBOUND(<u>x</u> , <u>n</u>)	Lower bound of array <u>x</u> in the <u>n</u> th dimension
PROD(<u>x</u>)	Product of the elements in array <u>x</u>
SUM(<u>x</u>)	Sum of the elements in array <u>x</u>

The Storage-handling Built-in Functions

These built-in functions are related to the allocation and use of storage, particularly CONTROLLED and BASED storage, as well as storage allocated in an AREA.

<u>Function</u>	<u>Returns</u>
ADDR(x)	POINTER address of <u>x</u>
ALLOCATION(x)	The number of allocations of the CONTROLLED variable <u>x</u>
EMPTY()	An empty AREA value
NULL()	A null POINTER value
OFFSET(ptr, a)	The offset of POINTER <u>ptr</u> in AREA <u>a</u>
POINTER(o, a)	POINTER to storage at OFFSET <u>o</u> in AREA <u>a</u>
SIZE(v[, n])	The size of the storage occupied by variable <u>v</u> measured in unit <u>n</u>

The Condition-handling Built-in Functions

These built-in functions are discussed further in Chapter 13, PL/I CONDITION HANDLING. Use them in an on-unit to provide information on why the on-unit was invoked. ONCODE() returns an integer value, while all the others return a CHARACTER value.

<u>Function</u>	<u>Returns</u>
ONCHAR()	The invalid character causing a CONVERSION error
ONCODE()	The internal integer value of the error code
ONFIELD()	Bad input stream characters for GET DATA
ONFILE()	Name of file on which an input/output error has occurred
ONKEY()	Key value for keyed file error

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

<u>Function</u>	<u>Returns</u>
ONLOC()	Name of entry point to last invoked procedure
ONSOURCE()	Invalid CHARACTER string causing CONVERSION error

Miscellaneous Built-in Functions

The remaining built-in functions do not fit into any of the above classifications.

<u>Function</u>	<u>Returns</u>
DATE()	Current date as CHARACTER(6) value, 'yymmdd'
LINENO(f)	Current line number on PRINT file <u>f</u>
PAGENO(f)	Current page number on PRINT file <u>f</u>
TIME()	Current time of day as CHARACTER(9) value, 'hhmmssfff'
UNSPEC(x)	BIT string internal representation of variable <u>x</u>
VALID(x)	BIT(1) logical test for validity of string value of PICTURE variable <u>x</u>

COMPLETE LIST OF BUILT-IN FUNCTIONS

The following pages provide descriptions of each of the built-in functions supported by Prime PL/I.

In these pages, PI is a FLOAT variable equal to 3.14159.

The ABS Arithmetic Built-in Function

ABS(x) returns the absolute value of x.

Format: ABS(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ABS for each aggregate element. Below, assume that x is scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: If the mode of x is REAL, the data type returned by ABS is the same as the data type of x.

If the mode of x is COMPLEX, the base and scale of the data type returned by ABS are the same as for the data type of x, the mode of the data type returned by ABS is real. The precision returned is (p) for FLOAT or (p, q) for FIXED, computed as follows: let r be the number of digits in the precision of x, and s the scale factor, if any, of the data type of x. Let n equal the maximum number of digits for the base and scale of x. Then:

$$p = \text{MIN}(n, r + 1)$$

$$q = s$$

Operation: PL/I returns the absolute value of x.

If x is REAL, this is merely the value of x with the sign made positive.

If x is COMPLEX, ABS(x) returns:

$$\text{SQRT}(\text{REAL}(x)**2 + \text{IMAG}(x)**2)$$

Examples: The following chart illustrates the ABS built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	ABS(2.34)	+2.34
2	ABS(-2.34)	+2.34
3	ABS(0)	+0
4	ABS(3.0E0-4E0I)	+5.00E0

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Lines 1 through 3 in the table illustrate the use of ABS with REAL arguments. ABS makes the sign of the argument positive and returns that result.

Line 4 illustrates ABS with a COMPLEX argument. Notice that the precision of the result is one greater than the precision of the argument.

The ACOS Mathematical Built-in Function

The ACOS built-in function returns the arc cosine, in radians, of the argument.

Format: ACOS(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ACOS for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a scale of FLOAT, with the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

It is an error if the mode of x is COMPLEX.

Result Data Type: The data type returned by ACOS is the same as the data type of x.

Operation: The value of x must be such that

$$-1 \leq x \leq +1$$

It is an error if x is outside of this range.

The mathematical arc cosine function is a multiple-valued function. PL/I returns the value w equal to the arc cosine of x such that

$$0 \leq w \leq \text{PI}$$

Examples: ACOS(1.000E0) returns +0.000E0. ACOS(0.0000) returns the value 1.5707E0, which equals PI/2.

The ADD Arithmetic Built-in Function

The ADD built-in function returns the sum of its first two arguments in the specified precision.

Format: ADD(x, y, p) or ADD(x, y, p, q)

Arguments: If x and y are not both scalars, PL/I applies the general rule for aggregate arguments and computes ADD for each aggregate element. Below, assume that x and y are scalar.

PL/I converts x and y to the derived common base, scale, and mode of the data type of x and y, and to the converted precision of the data type of the argument. In the following, assume that x and y are the converted values.

For information on the arguments p and q, see the section Arguments That Specify Precision earlier in this chapter.

Result Data Type: The result data type is as described in the section Arguments That Specify Precision earlier in this chapter.

Operation: PL/I computes the value of $x + y$, and converts it to the result data type.

Examples: The following chart illustrates the ADD built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	ADD(3, 5, 1)	8
2	ADD(3, 5, 3)	8
3	ADD(3, 5, 3, 1)	8.0
4	ADD(3E0, 5.2, 4)	8.200E0
5	ADD(3E0, 5.2, 4, 1)	ERROR

Line 5 is invalid because q is specified where the derived common scale of the first two arguments is FLOAT.

The ADDR Storage-handling Built-in Function

The ADDR built-in function returns a POINTER address to the argument.

Format: ADDR(x)

Argument: The argument x must be a scalar variable reference.

Result Data Type: The ADDR built-in function returns a POINTER scalar value.

Operation: PL/I returns the POINTER address of x.

If the storage class of x is CONTROLLED, but x has no allocations, PL/I returns a null POINTER value.

The use of ADDR is discussed in Chapter 7.

The AFTER String-handling Built-in Function

The AFTER built-in function returns the portion of a string following a substring.

Format: AFTER(s, c)

Arguments: If s and c are not both scalar, PL/I applies the general rule for aggregate arguments and computes AFTER for each aggregate element. Below, assume that s and c are scalar.

PL/I converts s and c to their common derived string type (CHARACTER or BIT). In the following, assume that s and c are the converted values.

Result Data Type: The data type returned by AFTER is the same as the common derived string type of s and c.

Operation: PL/I computes AFTER(s, c) as follows:

1. If s is a null string, PL/I returns a null string.
2. If c is a null string, PL/I returns s.
3. If c is not a null string, and c is not a substring of s, PL/I returns a null string.
4. If c is not a null string but is a substring of s, PL/I returns the portion of string s that follows the leftmost occurrence of c in s. More precisely, AFTER(s, c) returns

SUBSTR(s, INDEX(s, c) + LENGTH(c)).

PL/I Reference Guide

Examples: The following chart illustrates the AFTER built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	AFTER('', 'ABC')	''
2	AFTER('ABC', '')	'ABC'
3	AFTER('ABC', 'X')	''
4	AFTER('ABC', 'B')	'C'
5	AFTER('ABC', 'BC')	''
6	AFTER('ABCDABCD', 'BC')	'DABCD'
7	AFTER('10110'B, '01'B)	'10'B

Lines 1 and 2 illustrate the degenerate cases, where one of the arguments is a null string.

In line 3, the second argument is not a subscript of the first argument, and so PL/I returns the null string.

In lines 4 through 7, the second argument is a substring of the first. In line 5, the substring 'BC' is right at the end of 'ABC', and so PL/I returns the null string. Line 6 illustrates the case where the second argument occurs more than once in the first argument; only the first occurrence matters. Line 7 illustrates AFTER with BIT arguments.

The ALLOCATION Storage-handling Built-in Function

The ALLOCATION built-in function returns the number of allocations of a CONTROLLED argument.

Abbreviation: ALLOCN for ALLOCATION

Format: ALLOCATION(v)

Arguments: The argument v must be a level-1 variable with the CONTROLLED storage class.

Result Data Type: ALLOCATION returns the integer data type FIXED BINARY(15, 0) REAL.

Operation: If v has no allocations, PL/I returns the value 0; otherwise, PL/I returns the number of allocations.

Example: Consider the following program segment:

```

DECLARE ARC(1000) CONTROLLED;
PUT LIST(ALLOCATION(ARC));
ALLOCATE ARC;
PUT LIST(ALLOCATION(ARC));
ALLOCATE ARC;
PUT LIST(ALLOCATION(ARC));
FREE ARC;
FREE ARC;
PUT LIST(ALLOCATION(ARC));

```

The four PUT statements print the values 0, 1, 2, and 0, respectively.

The ASIN Mathematical Built-in Function

The ASIN built-in function returns the arc sine, in radians, of the argument.

Format: ASIN(x)

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ASIN for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of ASIN equals the data type of x.

Operation: The value of x must be in the range defined by

$$-1 \leq x \leq +1$$

ASIN returns the value w equal to the arc sine of x, such that

$$-\pi/2 \leq w \leq \pi/2$$

PL/I Reference Guide

Example: `ASIN(1.0000E0)` returns `+1.57E0`, which equals `PI/2`.
`ASIN(0.0000)` returns the value `+0.0000E0`.

ATAN Mathematical Built-in Function

The `ATAN` built-in function returns the arc tangent, in radians, of the argument, or of the quotient of two arguments.

Format: `ATAN(x)` or `ATAN(x, y)`

Arguments: If `x` is not a scalar, or if `y` is specified and is not a scalar, PL/I applies the general rule for aggregate arguments and computes `ATAN` for each aggregate element. Below, assume that `x` is scalar, and that `y`, if specified, is scalar.

PL/I converts `x` and `y` (if specified) to a scale of `FLOAT`, with the derived common base and mode of the data type of `x` (and `y`, if specified), and to the converted precision or precisions of the data types of the arguments. In the following, assume that `x` and `y` are the converted values.

If `y` is specified, the derived common mode must be `REAL`; if it is `COMPLEX`, the reference is illegal.

Result Data Type: The base, scale, and mode of the data type returned by `ATAN` are the same as for the data type of `x`. If `y` is not specified, the precision of the data type returned by `ATAN` is the same as for `x`. If `y` is specified, the precision of the data type returned by `ATAN` equals the maximum of the precisions of the data types of `x` and `y`.

Operation: PL/I performs the following steps:

1. If `y` is specified, it is illegal for `y` and `x` both to be 0. PL/I returns a value `w`, equal to the arc tangent, in radians, of `y/x`, such that

if `y >= 0`, then $0 \leq w \leq \text{PI}$, and

if `y < 0`, then $-\text{PI} < w < 0$.

2. If `y` is not specified, and `x` is `REAL`, PL/I returns a value `w`, equal to the arc tangent in radians of `x`, such that

$-\text{PI}/2 < w < \text{PI}/2$

3. If y is not specified, and if x is COMPLEX, it is illegal for x to have either of the values $1i$ or $-1i$. Otherwise, PI/I returns a value w , equal to the arc tangent in radians of x , such that

$$-PI < REAL(w) \leq PI$$

Discussion and Examples: The major purpose of ATAN with two arguments is to allow you to compute the arc tangent of a value near infinity. ATAN(x , y) returns the arc tangent of y/x in the quadrant of the coordinates of (x , y) as shown in Figure 14-1.

In particular, if $x = 0$, so that y/x is infinite, then ATAN(x , y) is defined and equals $PI/2$ if $y > 0$, and equals $-PI/2$ if $y < 0$.

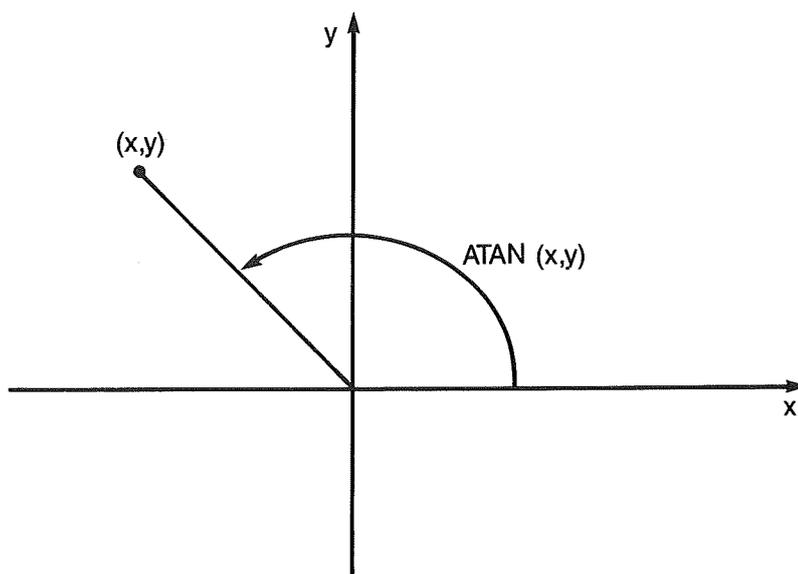


Figure 14-1
ATAN With Two Arguments

The following chart illustrates the ATAN built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	ATAN(0.0000)	0.0000E0
2	ATAN(1.0000,0)	1.5708E+00
3	ATAN(0,0)	ERROR
4	ATAN(0 + 0I, 1)	ERROR
5	ATAN(0 + 1I)	ERROR
6	ATAN(-0.6878 + 0.1584I)	1.8742E0 + 5.4321E -1I

Line number 1 computes the value of arc tangent 0, and returns 0. In line number 2, ATAN(1, 0) computes the arc tangent of 1/0. Even though PL/I is computing the arc tangent of an infinite value, ATAN(1,0) returns the value of PI/2.

Lines 3 through 5 illustrate the important error cases, and line 6 illustrates ATAN with a valid COMPLEX argument.

The ATAND Mathematical Built-in Function

The ATAND built-in function returns the arc tangent, in degrees, of the argument, or of the quotient of two arguments.

Format: ATAND(x) or ATAND(x,y)

ATAND is like ATAN, except that PL/I returns the arc tangent measured in degrees rather than in radians. ATAND does not permit COMPLEX arguments.

To compute the value of ATAND, PL/I computes the value of ATAN with the same argument or arguments and multiplies the result by 180/PI.

Example: ATAND(1.0000,0) returns 9E+01.

The ATANH Mathematical Built-in Function

The ATANH built-in function returns the inverse hyperbolic tangent of the argument.

Format: ATANH(x)

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ATANH for each aggregate element. Below, assume that x is scalar.

PL/I converts x to a scale of FLOAT, with the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by ATANH is the same as the data type of x.

Operation: If the mode of the data type of x is REAL, it is required that

$$-1 < x < +1$$

The reference is illegal otherwise.

If the mode of the data type of x is COMPLEX, the reference is illegal if x equals either +1 or -1.

PL/I returns the inverse hyperbolic tangent of x.

Example: ATANH(1.0000,0) returns 9E+01.

The BEFORE String-handling Built-in Function

The BEFORE built-in function returns the portion of the first argument that comes before the substring specified by the second argument.

Format: BEFORE(s, c)

Argument: If s and c are not both scalar, PL/I applies the general rule for aggregate arguments and computes BEFORE for each aggregate element. Below, assume that s and c are scalar.

PL/I converts s and c to the common derived string type (CHARACTER or BIT) of the data types of s and c. In the following, assume that s and c are the converted values.

Result Data Type: The data type of the value returned by BEFORE is the same as the data type of s.

PL/I Reference Guide

Operation: PL/I performs the following steps:

1. If either s or c is a null string, PL/I returns a null string.
2. If c is not a null string, and c is not a substring of s, PL/I returns the string s.
3. If c is not a null string but is a substring of s, PL/I returns the portion of string s that occurs to the left of the leftmost occurrence of c in s. More precisely, BEFORE(s, c) returns the following value:

SUBSTR(s, 1, INDEX(s, c) -1)

Examples: The following chart illustrates the BEFORE built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	BEFORE('', 'ABC')	''
2	BEFORE('ABC', '')	''
3	BEFORE('ABC', 'X')	'ABC'
4	BEFORE('ABC', 'B')	'A'
5	BEFORE('ABC', 'AB')	''
6	BEFORE('ABCDABCD', 'BC')	'A'
7	BEFORE('10110'B, '01'B)	'1'B

Lines 1 and 2 illustrate that BEFORE returns a null string whenever either argument is a null string.

In line 3, 'X' is not a substring of 'ABC', and so PL/I returns the first argument, 'ABC'.

Lines 4 through 6 illustrate the case where the second argument is a substring of the first argument. In line 4, 'B' is preceded by 'A' in 'ABC'. In line 5, nothing precedes 'AB' in 'ABC', and so BEFORE returns the null string. Line 6 illustrates the fact that if the second argument occurs more than once as a substring of the first argument, only the leftmost occurrence matters. Line 7 illustrates BEFORE with BIT arguments.

The BINARY Arithmetic Built-in Function

The BINARY built-in function converts the base of the argument to BINARY.

Abbreviation: BIN for BINARY

BUILT-IN FUNCTIONS AND PSEUDOVARIBLES

Format: BINARY(x) or BINARY(x, p) or BINARY(x, p, q)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes BINARY for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

For information on the arguments p and q, if specified, see the section Arguments That Specify Precision, near the beginning of this chapter.

Result Data Type: The base of the data type returned by BINARY is BINARY, and the scale and mode are the same as for the data type of x. The precision of the data type returned by BINARY is determined as described in the section Arguments That Specify Precision earlier in this chapter.

Operation: PL/I converts x to the result data type and returns that value.

Examples: The following chart illustrates the BINARY built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	BINARY(5)	+0101B
2	BINARY(5, 3)	+101B
3	BINARY(5, 5, 1)	+0101.0B
4	BINARY(5E0)	+1.010E2B
5	BINARY(5E0, 3)	+1.01E2B
6	BINARY(5E0, 5, 1)	ERROR

Lines 1 through 3 illustrate a FIXED conversion from DECIMAL to BINARY, while lines 4 and 5 illustrate a FLOAT conversion.

The reference in line 6 is illegal because q may not be specified for a FLOAT data type.

The BIT String-handling Built-in Function

The BIT built-in function converts the argument to a BIT string.

Format: BIT(x) or BIT(x, n)

Arguments: BIT takes any numeric data type, or a character string whose elements are all numbers (all 1's and 0's). If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes BIT for each aggregate element. Below, assume that x is a scalar.

The argument n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that n is the integer value.

Result Data Type: The data type of the value returned by BIT is BIT.

Operation: PL/I converts x to the data type BIT.

If n is specified, PL/I either pads or truncates the result of the conversion, so that the length of the resulting BIT string is n. (Note: A negative value of n is an illegal reference.) If n is not specified, the resulting length is the length obtained on conversion.

PL/I returns the BIT string.

Examples: The following chart illustrates the BIT built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	BIT(5)	'0101'B
2	BIT(5,3)	'010'B
3	BIT(5E0)	'0101'B
4	BIT(5E0,3)	'010'B
5	BIT(5.1E-3)	'0000000'B
6	BIT(-5,3)	'101'B

The BOOL String-handling Built-in Function

The BOOL built-in function permits you to perform any of the 16 logical operations on two BIT string values. You can use BOOL to get any logical operation, including AND, OR, exclusive OR, equals, and implies.

Format: BOOL(x, y, c)

Arguments: If x and y are not both scalar, PL/I applies the general rule for aggregate arguments and computes BOOL for each aggregate element. Below, assume that x and y are scalar.

PL/I converts x and y to BIT. If these strings have different lengths, PL/I pads the shorter one with 0-bits, to the length of the longer string. In the following, assume that x and y are the converted and padded BIT strings. The result is that x and y are two BIT strings of the same length.

The third argument, c, must be a scalar. PL/I converts c to BIT with a length of 4. In the following, assume that c is the converted value.

Result Data Type: The data type returned by BOOL is BIT. The length of the string returned by BOOL is equal to the common length of x and y.

Operation: Let b1, b2, b3, and b4 be the four bits in c. PL/I creates a new BIT string, whose length is the same as the common length of x and y, as follows: the value of the bit in the result BIT string is determined by the bits in the corresponding position in the arguments x and y, as shown in Table 14-1.

Table 14-1
Value of Result String

Bit in x	Bit in y	Result Corresponds to Bit in c
0	0	b1
0	1	b2
1	0	b3
1	1	b4

PL/I returns the resulting BIT string as the value of BOOL.

Discussion: There are 16 possible values of the BIT(4) string in the third argument to BOOL, and these correspond to the 16 logical functions on two truth values. The results of BOOL can be expressed in terms of other logical operators, as shown in Table 14-2.

Table 14-2
Value of BOOL

Value of c	Equivalent Logical Expression for BOOL(x, y, c)	Logical Description
'0000'B	All bits '0'B	False
'0001'B	$x \& y$	And
'0010'B	$x \& \hat{y}$	Does not imply
'0011'B	x	
'0100'B	$\hat{x} \& y$	
'0101'B	y	
'0110'B	$(x \& \hat{y}) \mid (\hat{x} \& y)$	Exclusive or
'0111'B	$x \mid y$	Inclusive or
'1000'B	$\hat{x} \& \hat{y}$	
'1001'B	$(x \& y) \mid (\hat{x} \& \hat{y})$	Equivalent
'1010'B	\hat{y}	Not
'1011'B	$x \mid \hat{y}$	Reverse implies
'1100'B	\hat{x}	Not
'1101'B	$\hat{x} \mid y$	Implies
'1110'B	$\hat{x} \mid \hat{y}$	
'1111'B	All bits '1'B	True

Examples: The following chart illustrates the BOOL built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	BOOL('01101'B, '11100'B, '1101'B)	'11110'B
2	BOOL('01101'B, '1'B, '1101'B)	'10010'B
3	BOOL('01101'B, '11100'B, '0110'B)	'10001'B
4	BOOL('01101'B, '1'B, '0110'B)	'11101'B

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

The third argument in lines 1 and 2 of this chart is '1101'B. Therefore, these two references to `BOOL` are equivalent to applying the logical operation implies to the first two arguments. In lines 3 and 4, the third argument is '0110'B, yielding the logical operation exclusive OR.

In lines 2 and 4, the second argument is shorter than the first argument, and so the second argument is padded with 0-bits to get '10000'B.

The BYTE String-handling Built-in Function (Prime Extension)

The `BYTE` built-in function converts a numeric argument to its character equivalent, according to its position in the collating sequence.

WARNING

`BYTE` is not an ANS PL/I function and is not available in other implementations of PL/I.

Format: `BYTE(x)`

Arguments: The argument x must be a scalar.

PL/I converts x to the integer data type `FIXED BINARY(15, 0) REAL`. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by `BYTE` is `CHARACTER`. The length of the string returned by `BYTE` is 1.

Operation: PL/I takes the integer value of x, forms a character out of the rightmost seven bits in that value, and returns that character.

The value that PL/I returns is equivalent to

`SUBSTR(COLLATE(), x + 1, 1)`.

Example: `BYTE(65)` returns 'A'.

The CEIL Arithmetic Built-in Function

The CEIL built-in function takes a noninteger argument and returns the next higher integer.

Format: CEIL(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes CEIL for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The base, scale, and mode of the data type returned by CEIL are the same as those of the data type of x.

If the scale of the data type of x is FLOAT, the precision of the data type of the value returned by CEIL is the same as the precision of the data type of x.

If the scale of the data type of x is FIXED, and the precision of the data type of x is (r, s), the precision of the data type of the value returned by CEIL is (p, 0), with a scale factor of 0, where

$$p = \text{MIN}(N, \text{MAX}(r - s + 1, 1))$$

and n is the maximum number of digits permitted for a scale of FIXED with a base of x. (n is 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I returns the smallest integer that is greater than or equal to x.

Examples: The following chart illustrates the CEIL built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	CEIL(2.3)	+03
2	CEIL(-2.3)	-02
3	CEIL(2)	+02
4	CEIL(-2)	-02
5	CEIL(2.3E0)	+3.0E0

Lines 1 and 2 illustrate CEIL with noninteger arguments, and lines 3 and 4 illustrate integer arguments. Line 5 illustrates a FLOAT noninteger argument.

The CHARACTER String-handling Built-in Function

The CHARACTER built-in function converts the argument to a CHARACTER string.

Format: CHARACTER(x) or CHARACTER(x, n)

Abbreviation: CHAR for CHARACTER

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes CHARACTER for each aggregate element. Below, assume that x is a scalar.

If the argument n is specified, n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that n is an integer value.

Result Data Type: The data type of the value returned by CHARACTER is CHARACTER.

Operation: PL/I converts x to the CHARACTER data type.

If n is specified, PL/I either pads or truncates the CHARACTER result of the conversion so that the length of the resulting string is n. (Note: A negative value of n is an illegal reference.) If n is not specified, the resulting length is the length obtained upon conversion.

PL/I returns this CHARACTER string.

PL/I Reference Guide

Example:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	CHARACTER(5)	'5'
2	CHARACTER(5.2)	'5.2'
3	CHARACTER(5.2,5)	'5.2 '

The COLLATE String-handling Built-in Function

The COLLATE built-in function returns the entire collating sequence.

Format: COLLATE()

Arguments: None

Result Data Type: The data type of the value returned by COLLATE is CHARACTER. The length of the string returned by COLLATE is 128 when the program is compiled with the -NO_EXTENDED_CHARACTER_SET option (the default), and 256 when the program is compiled with the -EXTENDED_CHARACTER_SET option.

Operation: PL/I returns a CHARACTER(256) string containing the entire collating sequence.

The COMPLEX Mathematical Built-in Function

The COMPLEX built-in function converts two REAL arguments to a COMPLEX value whose real and imaginary parts are the two arguments, respectively.

Format: COMPLEX(x, y)

Abbreviation: CPLX for COMPLEX

Arguments: If x and y are not both scalar, PL/I applies the general rule for aggregate arguments and computes COMPLEX for each aggregate element. Below, assume that x and y are scalar.

PL/I converts x and y to the common derived base, scale, and mode of the data types of x and y, and to the converted precision of each data type. In the following, assume that x and y are the converted values.

The derived common mode must be REAL; if the derived common mode is COMPLEX, the reference is illegal.

Result Data Type: The data type of the result returned by the COMPLEX built-in function has a mode of COMPLEX, and a base and scale equal to the common base and scale of the data types of \underline{x} and \underline{y} .

The precision of the data type returned by COMPLEX is determined as follows:

- If the common scale of the data types of \underline{x} and \underline{y} is FLOAT, the precision of the data type returned by COMPLEX equals the maximum of the precisions of the data types of \underline{x} and \underline{y} .
- If the common scale of the data types of \underline{x} and \underline{y} is FIXED, suppose that the data type of \underline{x} has a precision of (r, s) , and the precision of the data type of \underline{y} has precision (t, u) . Then the precision of the value returned by COMPLEX is (p, q) , where

$$p = \text{MIN}(n, \text{MAX}(r - s, t - u) + \text{MAX}(s, u))$$

$$q = \text{MAX}(s, u)$$

where n is the maximum number of digits permitted for a data type with a scale of FIXED and the common base of \underline{x} and \underline{y} . (n is 31 for FIXED BINARY and 14 for FIXED DECIMAL.)

Operation: PL/I forms a COMPLEX value whose real part is \underline{x} and whose imaginary part is \underline{y} . PL/I returns this COMPLEX value.

Example: COMPLEX(2, 3) returns the value $2 + 3I$.

The CONJG Mathematical Built-in Function

The CONJG built-in function returns the complex conjugate of a complex argument.

Format: CONJG(\underline{x})

Arguments: If \underline{x} is not a scalar, PL/I applies the general rule for aggregate arguments and computes CONJG for each aggregate element. Below, assume that \underline{x} is a scalar.

PL/I converts \underline{x} to a data type with a mode of COMPLEX and with the derived base and scale of the data type of \underline{x} and the converted

precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by CONJG has a mode of COMPLEX and has the same base, scale, and precision as the data type of x.

Operation: PL/I returns the complex conjugate of x. The conjugate of $a + bi$ is $a - bi$.

Example: The reference CONJG(2 + 3I) returns the value 2 - 3I.

The COPY String-handling Built-in Function

The COPY built-in function concatenates a string with itself a specified number of times.

Format: COPY(s, n)

Arguments: If s is not a scalar, PL/I applies the general rule for aggregate arguments and computes COPY for each aggregate element. Below, assume that s is a scalar.

PL/I converts s to the derived string type (CHARACTER or BIT) of the data type of x. In the following, assume that x is the converted value.

The argument n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that n is the integer value.

Result Data Type: The data type of the value returned by COPY is the same as the data type of x.

Operation: The reference is illegal if $n < 0$.

If $n = 0$, PL/I returns the null string.

If $n > 0$, PL/I returns a string of length $n * \text{LENGTH}(s)$, which contains n copies of the string s.

Examples: COPY('ABC', 3) returns 'ABCABCABC'. COPY('ABC', 0) returns the null string. COPY('1011'B, 2) returns '10111011'B.

The COS Mathematical Built-in Function

The COS built-in function returns the cosine of the argument, where the argument is given in radians.

Format: COS(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes COS for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by COS is the same as the data type of x.

Operation: PL/I returns the cosine of x, where x is an angle measured in radians.

Examples: COS(0) returns the value 1E0. COS(PI/2) returns 0.0000E0.

The COSD Mathematical Built-in Function

The COSD built-in function returns the cosine of the argument, where the argument is measured in degrees.

Format: COSD(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes COSD for each aggregate element. Below, assume that x is scalar.

PL/I converts x to the scale of FLOAT, the derived base and mode of the data type of x, and the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by COSD is the same as the data type of x.

Operation: PL/I returns the cosine of x, where x is an angle measured in degrees.

Examples: COSD(0) returns 1E0. COSD(90.000) returns 1.0000E0.

The COSH Mathematical Built-in Function

The COSH built-in function returns the hyperbolic cosine of the argument.

Format: COSH(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes COSH for each aggregate element. Below, assume that x is scalar.

PL/I converts x to the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by COSH is the same as the data type of x.

Operation: PL/I returns the hyperbolic cosine of x.

Examples: COSH(0) returns 1.E+00. COSH(90.0000) returns 8.5070E+37.

The DATE Built-in Function

The DATE built-in function returns the current date as a CHARACTER string, in the format 'yymmdd'.

Format: DATE()

Arguments: None

Result Data Type: The data type of the value returned by DATE is CHARACTER. The length of the string returned by DATE is 6.

Operation: PL/I returns a CHARACTER(6) value in the format 'yy \overline{mm} dd', where \overline{yy} represents the last two digits of the current year, \overline{mm} represents a two-digit value (01 to 12) for the current month, and \overline{dd} stands for the day of the month (01 to 31).

Example: On January 5, 1986, DATE() would return '860105'.

The DECAT String-handling Built-in Function

The DECAT built-in function breaks up a string as desired, according to the position of a substring.

Format: DECAT(s, c, t)

Arguments: If \underline{s} and \underline{c} are not both scalar, PL/I applies the general rule for aggregate arguments and computes DECAT for each aggregate element. Below, assume that \underline{s} and \underline{c} are scalar.

PL/I converts \underline{s} and \underline{c} to the common derived string type (CHARACTER or BIT) of the data types of \underline{s} and \underline{c} . In the following, assume that \underline{s} and \underline{c} are the converted values.

The argument \underline{t} must be a scalar. PL/I converts \underline{t} to a BIT string value of length 3. In the following, assume that \underline{t} is the converted value.

Result Data Type: The data type of the value returned by DECAT is the same as the data type of \underline{s} .

Operation: PL/I defines three strings, as follows:

$s_1 = \text{BEFORE}(s, c)$

$$s_2 = \left\{ \begin{array}{l} '' \text{ if } c \text{ is not a substring of } s \\ c \text{ if } c \text{ is a substring of } s \end{array} \right\}$$

$s_3 = \text{AFTER}(s, c)$

PL/I returns the result of concatenating together two or all of the strings \underline{s}_1 , \underline{s}_2 , and \underline{s}_3 determined by corresponding bit positions in the BIT(3) value \underline{t} . Specifically, PL/I returns a value obtained by concatenating together the intermediate strings specified in Table 14-3, depending upon the value of \underline{t} .

Discussion: DECAT works by breaking up the string s into three parts, s1, s2, and s3, such that s1 || s2 || s3 is the same as the string s.

When c is a substring of s, the three parts are the portion of s before c, c itself, and the portion of s after c, respectively. When c is not a substring of s, the three parts are s, a null string, and another null string.

The third argument, t, lets you choose which combination of these three parts PL/I should return as the value of DECAT.

Table 14-3
Values Returned for DECAT

Value of t	Value Returned for DECAT	Comment
'000'B	Null string	
'001'B	s3	Same as AFTER(s, c)
'010'B	s2	
'011'B	s2 s3	
'100'B	s1	Same as BEFORE(s, c)
'101'B	s1 s3	
'110'B	s1 s2	
'111'B	s1 s2 s3	Same as string s

Examples: The following chart illustrates the DECAT built-in functions:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	DECAT('ABC', 'B', '100'B)	'A'
2	DECAT('ABC', 'B', '010'B)	'B'
3	DECAT('ABC', 'B', '001'B)	'C'
4	DECAT('ABC', 'B', '101'B)	'AC'
5	DECAT('100101'B, '101'B, '101'B)	'100'B

In lines 1 through 4, the first two arguments are the same, and the second argument is a substring of the first. As a result, in each case, PL/I breaks up the first string into three parts, with s1 = 'A', s2 = 'B', and s3 = 'C'. In line 1, the third argument is '100'B, and so PL/I returns only s1, or 'A'. When the third argument is '100'B, DECAT is the same as BEFORE. In line number 2, the third argument is '010'B, and so PL/I returns only s2, or 'B'. In line 3, the third argument is '001'B, and so PL/I returns only s3, or 'C'. When the

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

third argument is '001'B, DECAT is the same as AFTER. In line number 4, the third argument contains two 1-bits, in the first and third position, and so PL/I returns s1 || s3, or 'AC'.

Line number 5 illustrates DECAT with BIT string arguments.

As a final illustration, suppose that C is a CHARACTER VARYING variable. Consider the following program segment:

```
DO WHILE(INDEX(C, ' ') > 0);  
C = DECAT(C, ' ', '101'B);  
END;
```

The DO loop iterates as long as the string C contains a blank character. The assignment statement in the DO loop removes the blank character from C. Therefore, this loop removes all blanks from C.

The DECIMAL Arithmetic Built-in Function

The DECIMAL built-in function converts the base of the argument to DECIMAL.

Format: DECIMAL(x) or DECIMAL(x, p) or DECIMAL(x, p, q)

Abbreviation: DEC for DECIMAL

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes DECIMAL for each aggregate element. Below, assume that x is scalar.

For information on arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The base of the data type returned by DECIMAL is DECIMAL, and the scale and mode of the data type returned by DECIMAL are the derived scale and mode of the argument x. The precision of the result is described in the section Arguments That Specify Precision near the beginning of this chapter.

Operation: PL/I converts x to the result data type, and returns that value.

Examples:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	DECIMAL(101B)	5
2	DECIMAL(5.2E+0)	5.2E+00
3	DECIMAL(5.2E+0,10,2)	ERROR

The DIMENSION Array-handling Built-in Function

The DIMENSION built-in function returns the dimension size of an array argument for a specified dimension number.

Format: DIMENSION(x, n)

Abbreviation: DIM for DIMENSION

Arguments: The argument x must be an array. The argument n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that n is the converted integer value.

Result Data Type: The value returned by the DIMENSION built-in function is an integer value with data type FIXED BINARY(15, 0) REAL.

Operation: The value of n must be greater than or equal to 1, and less than or equal to the number of dimensions in the array x; if n is outside of this range, the reference is illegal.

PL/I computes the dimension size for the subscript position specified by n. The dimension size equals

$$(\text{upper bound}) - (\text{lower bound}) + 1$$

Example: Suppose the array A is declared as follows:

```
DECLARE A(10, 2:6);
```

In this case, the reference DIMENSION(A, 1) returns the value 10, since the dimension size in the first dimension is 10. The reference DIMENSION(A, 2) returns the integer value 5, since the dimension size in the second dimension is (6 - 2 + 1), or 5.

The DIVIDE Arithmetic Built-in Function

The DIVIDE built-in function returns the quotient of two values in the specified precision.

Format: DIVIDE(x, y, p) or DIVIDE(x, y, p, q)

Arguments: If x and y are not both scalar, PL/I applies the general rule for aggregate arguments and computes DIVIDE for each aggregate element. Below, assume that x and y are scalar.

PL/I converts x and y to the common derived base, scale, and mode of the data types of x and y, and to the converted precision of the data type of the respective argument. In the following, assume that x and y are the converted values.

For information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The data type of the result returned by DIVIDE is as described in the section Arguments That Specify Precision near the beginning of this chapter.

Operation: PL/I computes the value of x/y and converts it to the result data type.

Examples: The following chart illustrates the DIVIDE built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	DIVIDE(8,2,1)	4
2	DIVIDE(8,2,3)	4
3	DIVIDE(8,2,3,1)	4.0
4	DIVIDE(8E0,2.0,4)	4.000E+00
5	DIVIDE(8E0,2.0,4,1)	ERROR

In lines 1 through 3, the common derived base, scale, and mode are FIXED DECIMAL REAL, and the quotient is 4. The value returned depends upon the precision specified by the arguments p and q.

In line 4, the common derived scale is FLOAT, and the precision is 4.

Line 5 illustrates an invalid reference, because the argument q is specified when the derived scale is FLOAT.

The DOT Array-handling Built-in Function

The DOT built-in function returns the dot product of the two one-dimensional arrays.

Format: DOT(x, y) or DOT(x, y, p) or DOT(x, y, p, q)

Arguments: The arguments x and y must be one-dimensional arrays, with identical dimension bounds.

PL/I converts the arrays x and y to new arrays with the derived base, scale, and mode of the data types of x and y, and to the converted precision of the respective data types of the arguments. In the following, assume that x and y are the converted array values.

For information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

If you do not specify the arguments p and q, the scale of x and y must be FLOAT; if the derived scale is FIXED, and you do not specify the arguments p and q, the reference is illegal.

Result Data Type: The value returned by the DOT built-in function is a scalar, and has a data type as specified in the section Arguments That Specify Precision earlier in this chapter.

Operation: PL/I computes the dot product of the two arrays x and y, as follows: let l equal the common lower bound of the arrays x and y, and let u equal the common upper bound. PL/I computes the following value:

$$w = \sum_{k=l}^u x(k)y(k)$$

PL/I returns the value w. Note that a dot product, or scalar product, is the result of multiplication.

Example: Suppose that the arrays A and B are declared as follows:

```
DECLARE A(3) FLOAT DECIMAL(3) INITIAL(2,8,9);
```

```
DECLARE B(3) FLOAT DECIMAL(3) INITIAL(1,4,0);
```

then PL/I computes the value of DOT(A, B) by computing the value

$$2 * 1 + 8 * 4 + 9 * 0$$

and returns the value 3.40E+01.

The EMPTY Storage-handling Built-in Function

The EMPTY built-in function returns an empty AREA value.

Format: EMPTY()

Arguments: None

Result Data Type: The value returned by the EMPTY built-in function is a scalar with the AREA data type.

Operation: PL/I returns an empty AREA value.

Example:

```
DECLARE A AREA;
A = EMPTY();
```

The assignment statement shown above can be used at any time to free all allocations within the area A.

The ERF Mathematical Built-in Function

The ERF built-in function returns the error function of the argument.

Format: ERF(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ERF for each aggregate element. Below, assume that x is a scalar.

PL/I Reference Guide

PL/I converts x to the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the result returned by ERF is the same as the data type of x.

Operation: PL/I computes the following value:

$$(2/Pi) \int_0^x e^{-t^2} dt$$

PL/I returns this value.

The ERFC Mathematical Built-in Function

The ERFC built-in function returns the complement of the error function of the argument.

Format: ERFC(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ERFC for each aggregate element. Below, assume that x is scalar.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by ERFC is the same as the data type of x.

Operation: PL/I computes the value

$1 - \text{ERF}(x)$

and returns that value.

The EVERY String-handling Built-in Function

The EVERY built-in function tests whether all bits in a BIT scalar or aggregate are 1-bits and returns a single bit to indicate the result.

Format: EVERY(x)

Argument: The argument x must be either a scalar with the BIT data type, or an aggregate each of whose scalar elements has the BIT data type. (If x is not a scalar, it does not follow the general rule for aggregate arguments.)

Result Data Type: The data type of the value returned by EVERY is the logical data type BIT. The length of the string returned by EVERY is 1.

Operation: If x, or any scalar element of the aggregate x, contains a 0-bit, PL/I returns '0'B; otherwise, PL/I returns '1'B.

As a consequence of this rule, if the scalar x is a null string, or if all the scalar elements of the aggregate x are null strings, PL/I returns '1'B.

Examples: EVERY('11101'B) returns the value '0'B, since the argument contains a 0-bit. On the other hand, EVERY('111111'B) returns the value '1'B, since the argument contains only 1-bits. EVERY(''B) also returns '0'B.

As a further example, consider the following statements:

```
DECLARE A(10);
...
IF EVERY(A > 0) THEN . . .
```

In this example, the THEN clause of the IF statement is taken if EVERY(A > 0) is considered true; PL/I considers it true provided that every element of A is positive.

The EXP Mathematical Built-in Function

The EXP built-in function takes an argument x and calculates e (the base of the natural logarithm) to the power x.

Format: EXP(x)

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments, and computes EXP for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by EXP is the same as the data type of x.

Operation: PL/I computes the value of e^x , where e is the base of the natural logarithm (approximately 2.718), and returns that value.

Example: EXP(0) returns the value 1E0. EXP(1.000) returns the value 2.718E0.

The FIXED Arithmetic Built-in Function

The FIXED built-in function converts the argument to the scale of FIXED.

Format: FIXED(x, p) or FIXED(x, p, q)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes FIXED for each aggregate element. Below, assume that x is a scalar.

For information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

The data type of the value returned by the FIXED built-in function has a scale of FIXED, and the derived base and mode of the data type of the argument x.

The precision of the data type of the value returned by the FIXED built-in function is as described in the section Arguments That Specify Precision near the beginning of this chapter.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Operation: PL/I converts x to the result data type, and returns that value.

Examples: The FIXED built-in function is illustrated by the following chart:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	FIXED(3,4)	3
2	FIXED(3,5,1)	3.0
3	FIXED(2.824E0,2,1)	2.8

In each of these cases, the value returned has a precision as specified by the arguments p and q.

The FLOAT Arithmetic Built-in Function

The FLOAT built-in function converts the argument to a scale of FLOAT.

Format: FLOAT(x, p)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes FLOAT for each aggregate element. Below, assume that x is a scalar.

For information on the argument p, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The data type of the result returned by the FLOAT built-in function has a scale of FLOAT, and the derived base and mode of the argument x.

The precision of the data type of the value returned by FLOAT is as described in the section Arguments That Specify Precision.

Operation: PL/I converts x to the result data type and returns that value.

Examples: The following chart illustrates the FLOAT built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	FLOAT(3,4)	+3.000E0
2	FLOAT(3,5)	+3.0000E0
3	FLOAT(2.824E0,2)	+2.8E0

In each of these cases, PL/I converts the argument to FLOAT with a precision specified by the second argument. Notice that truncation takes place in line number 3.

The FLOOR Arithmetic Built-in Function

The FLOOR built-in function takes a noninteger argument and returns the next lower integer.

Format: FLOOR(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes FLOOR for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The base, scale, and mode of the data type of the result returned by FLOOR are the same as for the data type of x.

If the scale of the data type of x is FLOAT, the precision of the data type returned by FLOOR is the same as the precision of the data type of x.

If the scale of the data type of x is FIXED, and the precision of the data type of x is (r, s), the precision of the data type returned by FIXED is (p, 0), where the scale factor is 0, and where

$$p = \text{MIN}(n, \text{MAX}(r - s + 1, 1))$$

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

and n is the maximum number of digits permitted for a data type with a scale of `FIXED` and the base of the data type of x. (n is 31 for `FIXED BINARY`, and 14 for `FIXED DECIMAL`.)

Operation: `PL/I` returns the largest integer that is less than or equal to x.

Examples: The following chart illustrates the `FLOOR` built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	<code>FLOOR(2.7)</code>	+02
2	<code>FLOOR(-2.7)</code>	-03
3	<code>FLOOR(2)</code>	+02
4	<code>FLOOR(-2)</code>	-02
5	<code>FLOOR(2.3E0)</code>	+2.0E0

Lines 1 and 2 illustrate `FLOOR` with noninteger arguments, and lines 3 and 4 illustrate integer arguments. In lines 1 through 4, the argument is `FIXED`, while in line 5, the argument is a `FLOAT` noninteger value.

The HBOUND Array-handling Built-in Function

The `HBOUND` built-in function returns the upper bound of the array argument for the specified dimension number.

Format: `HBOUND(x, n)`

Arguments: The argument x must be an array. The argument n must be a scalar.

`PL/I` converts n to the integer data type `FIXED BINARY(15, 0) REAL`. In the following, assume that n is the converted integer value.

Result Data Type: The value returned by `HBOUND` is an integer value with data type `FIXED BINARY(15, 0) REAL`.

Operation: The value of n must be greater than or equal to 1 and less than or equal to the number of dimensions in the array x; if n is outside of this range, the reference is illegal.

`PL/I` returns the upper bound for the subscript position specified by n in the array x.

Example: Suppose the array A is declared as follows:

```
DECLARE A(10, 2:6);
```

In this case, HBOUND(A,1) returns the integer value 10, and HBOUND(A,2) returns 6.

The HIGH String-handling Built-in Function

The HIGH built-in function returns a CHARACTER string of specified length, each of whose characters is the highest character in the collating sequence.

Format: HIGH(n)

Arguments: The argument n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15,0) REAL. In the following, assume that n is the converted integer value.

Result Data Type: The data type of the value returned by HIGH is CHARACTER. The length of the string returned by HIGH is the integer n.

Operation: If the value of n is negative, the reference is illegal.

If $n = 0$, PL/I returns the null string.

If $n > 0$, PL/I returns a CHARACTER string of length n, each of whose characters equals the character that comes last in the collating sequence.

Examples: HIGH(0) returns the null string.

HIGH(30) returns a string of length 30 that is guaranteed to compare as greater than any other CHARACTER string of length 30.

The IMAG Mathematical Built-in Function

The IMAG built-in function returns the imaginary part of a COMPLEX argument.

Format: IMAG(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes IMAG for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with a mode of COMPLEX, to the derived base and scale of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is a converted value.

Result Data Type: The data type of the value returned by IMAG has a mode of REAL and has the same base, scale, and precision as the data type of x.

Operation: PL/I returns the imaginary part of x.

Example: The reference IMAG(2 + 3I) returns the value 3.

The INDEX String-handling Built-in Function

The INDEX built-in function returns the position of a specified substring within a string.

Format: INDEX(s, c)

Arguments: If s and c are not both scalars, PL/I applies the general rule for aggregate arguments and computes INDEX for each aggregate element. Below, assume that s and c are scalar.

PL/I converts s and c to the common derived string type (CHARACTER or BIT) of the data types of s and c. In the following, assume that s and c are the converted values.

Result Data Type: The value returned by the INDEX built-in function has the integer data type FIXED BINARY(15, 0) REAL.

PL/I Reference Guide

Operation: PL/I performs the following steps:

1. If either of the strings c or s is a null string, PL/I returns the integer value 0.
2. If c is not a substring of s, PL/I returns the integer value 0.
3. If c is not a null string, but is a substring of s, PL/I returns an integer value equal to the position of the leftmost occurrence of the substring c in s.

Examples: The following chart illustrates the INDEX built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	INDEX('ABCD','X')	0
2	INDEX('ABCDEF','DE')	4
3	INDEX('ABCDABCD','BCD')	2
4	INDEX('101101'B,'10'B)	1

In line 1, INDEX returns the integer value 0, since 'X' is not a substring of 'ABCD'.

In line 2, 'DE' is a substring of 'ABCDEF' in the fourth and fifth character positions, so INDEX returns the integer value 4. In line number 3, the string 'BCD' is a substring of 'ABCDABCD' in two different positions, but only the leftmost occurrence counts.

Line number 4 illustrates INDEX with BIT string arguments.

The LBOUND Array-handling Built-in Function

The LBOUND built-in function returns the lower bound of an array argument for the specified dimension number.

Format: LBOUND(x, n)

Arguments: The argument x must be an array. The argument n must be a scalar.

PL/I converts n to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that n is the converted integer value.

Result Data Type: The value returned by the LBOUND built-in function is an integer value with a data type of FIXED BINARY(15, 0) REAL.

Operation: The value of n must be greater than or equal to 1 and less than or equal to the number of dimensions in the array x. If n is out of range, the reference is illegal.

PL/I returns the lower bound for the subscript position specified by n.

Examples: Suppose the array A is declared as follows:

```
DECLARE A(10,2:6);
```

then LBOUND(A,1) returns the integer value 1, and LBOUND(A,2) returns 2.

The LENGTH String-handling Built-in Function

The LENGTH built-in function returns the length of a string argument.

Format: LENGTH(s)

Argument: If s is not a scalar, PL/I applies the general rule for aggregate arguments and computes LENGTH for each aggregate element. Below, assume that s is scalar.

PL/I converts s to the derived string type (CHARACTER or BIT) of the data type of s. In the following, assume that s is the converted value.

Result Data Type: The value returned by LENGTH is an integer value with data type FIXED BINARY(15, 0) REAL.

Operation: PL/I returns the number of characters or bits in the string s. If the string s is a null string, PL/I returns the integer value 0.

Examples: LENGTH('ABC') returns the value 3.

The LINENO Built-in Function

The LINENO built-in function returns the current output line number position on the current page of a PRINT OUTPUT STREAM file.

Format: LINENO(f)

Arguments: The argument f must be a scalar. The argument f must have the FILE data type.

Result Data Type: The value returned by the LINENO built-in function is an integer value with data type FIXED BINARY(15, 0) REAL.

Operation: The file f must be open with the STREAM OUTPUT PRINT attributes; otherwise the reference is illegal.

PL/I finds the current line number on the current page and returns that integer value.

The LOG Mathematical Built-in Function

The LOG built-in function returns the natural logarithm of the argument.

Format: LOG(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes LOG for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with the scale of FLOAT, the derived base and mode of the data type of x, and the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by LOG is the same as the data type of x.

Operation: PL/I performs the following steps:

1. If the mode of the data type of x is REAL, the value of x must be positive; otherwise, the reference is illegal. PL/I returns the natural logarithm of x.

2. If the mode of the data type of \underline{x} is COMPLEX, the value of \underline{x} may not be 0; if the value of \underline{x} is 0, the reference is illegal. PL/I returns a value \underline{w} of the natural logarithm of \underline{x} , such that

$$-\text{PI} < \text{IMAG}(w) \leq \text{PI}$$

Discussion: LOG computes the logarithm of its arguments to the base e , where $e = 2.718281828\dots$

The related built-in functions LOG2 and LOG10 compute the logarithm of their arguments to the bases 2 and 10, respectively.

Examples: LOG(1) returns 0E0. LOG(2.71828) returns 1.00000E0.

The LOG10 Mathematical Built-in Function

The LOG10 built-in function returns the common logarithm of the argument.

Format: LOG10(x)

Arguments: If \underline{x} is not a scalar, PL/I applies the general rule for aggregate arguments and computes LOG10 for each aggregate element. Below, assume that \underline{x} is a scalar.

PL/I converts \underline{x} to a data type with a scale of FLOAT, the derived base and mode of the data type of \underline{x} , and the converted precision of the data type of \underline{x} . In the following, assume that \underline{x} is the converted value.

The mode of the data type of \underline{x} must be REAL; if \underline{x} is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by LOG10 is the same as the data type of \underline{x} .

Operation: The value of \underline{x} must be positive; if \underline{x} is negative, the reference is illegal.

PL/I returns the common logarithm (logarithm to base 10) of \underline{x} .

Examples: LOG10(1) returns the value 0E0. LOG10(10) returns 1.0E0.

The LOG2 Mathematical Built-in Function

The LOG2 built-in function returns the logarithm of its argument to the base 2.

Format: LOG2(x)

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes LOG2 for each aggregate element. Below, assume that x is scalar.

PL/I converts x to the data type with the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by LOG2 is the same as the data type of x.

Operation: The value of x must be positive; if x is negative the reference is illegal.

PL/I returns the value of the logarithm of x to the base 2.

Examples: LOG2(1) returns 0E0. LOG2(2) returns 1E0.

The LOW String-handling Built-in Function

The LOW built-in function returns a CHARACTER string of specified length, each of whose characters is the lowest character in the collating sequence.

Format: LOW(n)

Arguments: The argument n must be a scalar. PL/I converts n to the integer data type FIXED BINARY(15,0) REAL. In the following, assume that n is the converted integer value.

Result Data Type: The data type of the value returned by the LOW built-in function is CHARACTER. The length of the string returned by LOW is given by the integer value n.

Operation: If the value of n is negative, the reference is illegal.

If $n = 0$, PL/I returns the null string.

If $n > 0$, PL/I returns a CHARACTER string of length n , each of whose characters is the character that comes first in the collating sequence.

Examples: LOW(0) returns the null string. LOW(30) returns a CHARACTER string of length 30 that is guaranteed to compare as less than any other string of length 30.

The MAX Arithmetic Built-in Function

The MAX built-in function computes the maximum of the values of its arguments.

Format: MAX(x_1, x_2, \dots, x_n). There must be one or more arguments.

Arguments: If the arguments are not all scalar, PL/I applies the general rule for aggregate arguments and computes MAX for each aggregate element. Below, assume that x_1, x_2, \dots, x_n are all scalar.

PL/I converts each argument to the common derived base, scale, and mode of the data types of the arguments, and to the converted precision of the data type of the respective argument. In the following, assume that x_1, x_2, \dots, x_n are the converted values.

Result Data Type: The data type of the value returned by MAX has the common derived base, scale, and mode of the arguments. The precision of the data type of the result returned by MAX is determined as follows:

- If the common derived scale of the arguments is FLOAT, the number of digits in the precision of the data type returned by MAX is the maximum of the precisions in the data types of x_1, x_2, \dots, x_n .
- If the common derived scale is FIXED, let $(p_1, q_1), (p_2, q_2), \dots, (p_n, q_n)$ be the precisions of the data types of x_1, x_2, \dots, x_n , respectively. Then the precision of the value returned by MAX is (p, q) , where

$$p = \text{MIN}(n, \text{MAX}(p_1 - q_1, p_2 - q_2, \dots, p_n - q_n))$$

$$- \text{MAX}(q_1, q_2, \dots, q_n)$$

$$q = \text{MAX}(q_1, q_2, \dots, q_n)$$

where n is the maximum number of digits permitted for a data type with a scale of FIXED and the common derived base of the arguments. (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I determines the maximum of the values of the arguments x1, x2, ..., xn. PL/I converts this value to the result data type and returns the converted result.

Example: MAX(-5, 8, 4) returns the value 8.

The MIN Arithmetic Built-in Function

The MIN built-in function returns the minimum of its arguments.

Format: MIN(x1, x2, ..., xn). There must be one or more arguments.

Arguments: If the arguments are not all scalar, PL/I applies the general rule for aggregate arguments and computes MIN for each aggregate element. Below, assume that x1, x2, ..., xn are all scalar.

PL/I converts each of the arguments to the common derived base, scale, and mode of the data types of the arguments, and to the converted precision of the data type of the respective argument. In the following, assume that x1, x2, ..., xn are the converted values.

The derived common mode of the data types of the arguments must be REAL; if it is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by the MIN built-in function has the common derived base, scale, and mode of the arguments. The precision of the data type of the value returned by the MIN built-in function is determined as follows:

- If the common derived scale is FLOAT, the precision of the data type returned by MIN equals the maximum of the precisions of the data types of x1, x2, ..., xn.

- If the common derived scale of the data types of the arguments is FIXED, let $(p_1, q_1), (p_2, q_2), \dots, (p_n, q_n)$ be the precisions of the data types of $\underline{x_1}, \underline{x_2}, \dots, \underline{x_n}$, respectively. Then the precision of the data type of the value returned by MIN is (p, q) , where

$$p = \text{MIN}(n, \text{MAX}(p_1, q_1, p_2 - q_2, \dots, p_n - q_n) \\ - \text{MAX}(q_1, q_2, \dots, q_n))$$

$$q = \text{MAX}(q_1, q_2, \dots, q_n)$$

where n is the maximum number of digits permitted for a data type with a scale of FIXED and the common derived base of the arguments. (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I computes the minimum of the values of the arguments $\underline{x_1}, \underline{x_2}, \dots, \underline{x_n}$. PL/I converts this value to the result data type, and returns the result of the conversion.

Example: MIN(-5, 8, 4) returns -5.

The MOD Arithmetic Built-in Function

The MOD built-in function returns the remainder resulting from the division of two arguments.

Format: MOD(x, y)

Arguments: If \underline{x} and \underline{y} are not both scalar, PL/I applies the general rule for aggregate arguments and computes MOD for each aggregate element. Below, assume that \underline{x} and \underline{y} are scalar.

PL/I converts each of \underline{x} and \underline{y} to the derived common base, scale, and mode of the data types of \underline{x} and \underline{y} , and to the converted precision of the respective argument. In the following, assume that \underline{x} and \underline{y} are the converted values.

The derived common mode of the arguments must be REAL; if either \underline{x} or \underline{y} is COMPLEX, the reference is illegal.

Result Data Type: The base, scale, and mode of the data type of the value returned by MOD are the same as the common derived base, scale, and mode of the data types of \underline{x} and \underline{y} .

The precision of the data type of the result returned by MOD is computed as follows:

- If the common derived scale is FLOAT, the precision of the data type of the value returned by MOD equals the maximum of the precisions of the data types of \underline{x} and \underline{y} .
- If the common derived scale is FIXED, let (r, s) and (t, u) be the precisions of the data types of \underline{x} and \underline{y} , respectively. Then the precision of the data type of the value returned by MOD is (p, q) , where

$$p = \text{MIN}(n, t - u + \text{MAX}(s, u))$$

$$q = \text{MAX}(s, u)$$

where n is the maximum number of digits permitted for a data type with the scale of FIXED and the common derived base of \underline{x} and \underline{y} . (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I computes the remainder obtained when \underline{x} is divided by \underline{y} . This computation is made according to the following rules:

1. If \underline{y} equals 0, PL/I returns the value \underline{x} .
2. If \underline{y} is not equal to 0, PL/I returns the value

$$\underline{x} - \underline{y} * \text{FLOOR}(\underline{x}/\underline{y})$$

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Examples: The following chart illustrates the MOD built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	MOD(11,4)	3
2	MOD(12,4)	0
3	MOD(13,4)	1
4	MOD(11.5,4)	3.5
5	MOD(12.26,4)	0.26
6	MOD(13.93,4)	1.93
7	MOD(-11,4)	1
8	MOD(-12,4)	0
9	MOD(-13,4)	3
10	MOD(11,-4)	-1
11	MOD(12,-4)	0
12	MOD(13,-4)	-3
13	MOD(5.3,2.5)	0.3
14	MOD(15.3E0,2.5)	3.00E-1

In line number 1, since 11 divided by 4 has a quotient of 2 with a remainder of 3, MOD(11,4) returns 3. PL/I actually computes this as follows:

$$\begin{aligned}
 &11 - 4 * \text{FLOOR}(11/4) \\
 &= 11 - 4 * \text{FLOOR}(2.75) \\
 &= 11 - 4 * 2 = 3.
 \end{aligned}$$

In line number 2, 12 divided by 4 has no remainder, so MOD(12,4) returns 0. In line number 3, 13 divided by 4 has a quotient of 3 and a remainder of 1, so MOD(13,4) returns the value 1.

Lines 4 through 6 illustrate a noninteger numerator or first argument to MOD. In line 4, 11.5 divided by 4 gives a quotient of 2, with a remainder of 3.5, so MOD(11.5,4) returns 3.5. Lines 5 and 6 are similar.

Lines 7 through 12 illustrate the fact that if the value returned by MOD is not 0, it has the same sign as the denominator of the division (second argument to MOD); that is, if the denominator is positive, the value returned is 0 or positive, and if the denominator is negative, the value returned is 0 or negative. In line 7, PL/I computes the value of MOD(-11,4) as follows:

$$\begin{aligned}
 &-11 - 4 * \text{FLOOR}(-11/4) \\
 &= -11 - 4 * \text{FLOOR}(-2.75) \\
 &= -11 - 4 * (-3) = 1
 \end{aligned}$$

Similarly, in line 10, PL/I computes the value of MOD(11,-4) as follows:

$$\begin{aligned} & -11 - (-4) * \text{FLOOR}(11/-4) \\ & = 11 - (-4) * \text{FLOOR}(-2.75) \\ & = 11 - (-4) * (-3) = -1 \end{aligned}$$

Lines 13 and 14 illustrate what happens when the denominator (second argument to MOD) is not an integer. For line 13, when 5.3 is divided by 2.5, the quotient is 2 and the remainder is 0.3, so MOD(5.3,2.5) returns the value 0.3.

The MULTIPLY Arithmetic Built-in Function

The MULTIPLY built-in function returns the product of two values in the specified precision.

Format: MULTIPLY(x, y, p) or MULTIPLY(x, y, p, q)

Arguments: If x and y are not both scalar, PL/I applies the general rule for aggregate arguments and computes MULTIPLY for each aggregate element. Below, assume that x and y are scalars.

PL/I converts each of x and y to the derived base, scale, and mode of the data types of x and y, and to the converted precision of the respective data type. In the following, assume that x and y are the converted values.

For information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The data type of the value returned by MULTIPLY is as described in the section Arguments That Specify Precision near the beginning of this chapter.

Operation: PL/I computes the value of $x * y$ and converts it to the result data type.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Examples: The following chart illustrates the MULTIPLY built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	MULTIPLY(2,4,1)	8
2	MULTIPLY(2,4,2)	08
3	MULTIPLY(2,4,5,1)	0008.0

In all three cases, the first two arguments are 2 and 4. The function returns the product of these two values in the various precisions specified by the third and, possibly, fourth arguments.

The NULL Storage-handling Built-in Function

The NULL built-in function returns a null POINTER value.

Format: NULL()

Arguments: None

Result Data Type: The data type of the value returned by NULL is POINTER.

Operation: PL/I returns a null POINTER value. The NULL built-in function is discussed in Chapter 7.

The OFFSET Storage-handling Built-in Function

The OFFSET built-in function converts a POINTER value to an OFFSET value within a specified area.

Format: OFFSET(ptr, a)

Arguments: If ptr is not a scalar, PL/I applies the general rule for aggregate arguments and computes OFFSET for each aggregate element. Below, assume that ptr is a scalar.

The argument ptr must have the POINTER data type. The argument a must be a scalar and must have the AREA data type.

Result Data Type: The data type of the value returned by OFFSET is OFFSET.

Operation: PL/I converts the POINTER value ptr to an OFFSET value within the area specified by a.

The ONCHAR Condition-handling Built-in Function

The ONCHAR built-in function returns the invalid character that caused PL/I to raise the CONVERSION condition.

Format: ONCHAR()

Arguments: None

Result Data Type: The data type of the value returned by ONCHAR is CHARACTER. The length of the string returned by ONCHAR is 1.

Operation: Use ONCHAR() in a CONVERSION or ERROR on-unit invoked as the result of an error in attempting to convert a CHARACTER string to some other data type. PL/I returns the invalid character that gave rise to the conversion error.

If you use ONCHAR in some other context, or if the CONVERSION or ERROR on-unit is invoked with a SIGNAL statement, PL/I returns a CHARACTER(1) value containing a blank character.

For more information on the ONCHAR built-in function, see Chapter 13.

The ONCODE Condition-handling Built-in Function

The ONCODE built-in function returns an integer error code that you can use in an on-unit to determine which error caused the on-unit to be raised.

Format: ONCODE()

Arguments: None

Result Data Type: The value returned by the ONCODE built-in function has the integer data type FIXED BINARY(15).

Operation: Use ONCODE() in an on-unit invoked as the result of an error. PL/I returns an integer value indicating the type of error causing the on-unit to be invoked.

For more information on the ONCODE built-in function, see Chapter 13. The meanings of the error codes returned are given in Appendix F.

The ONFIELD Condition-handling Built-in Function

The ONFIELD built-in function returns the input stream characters that caused the NAME condition to be raised from a GET DATA statement.

Format: ONFIELD()

Arguments: None

Result Data Type: The data type of the value returned by ONFIELD is CHARACTER.

Operation: Use ONFIELD() in a NAME or ERROR on-unit invoked as the result of an error in a GET DATA statement. PL/I returns the string of characters in the input stream that gave rise to the error.

If you use ONFIELD in some other context, or if the NAME or ERROR on-unit is invoked with the SIGNAL statement, ONFIELD returns a null string.

For more information on the ONFIELD built-in function, see Chapter 13.

The ONFILE Condition-handling Built-in Function

The ONFILE built-in function returns the name of the FILE constant for which an input/output error on-unit was raised.

Format: ONFILE()

Arguments: None

Result Data Type: The data type of the value returned by ONFILE is CHARACTER.

Operation: Use ONFILE() in an on-unit meant to handle an input/output error. PL/I returns the name of the file constant on which the error occurred.

If you use ONFILE() in some other context, or if the on-unit was invoked with a SIGNAL statement, ONFILE returns a null string.

For more information on the ONFILE built-in function, see Chapter 13.

The ONKEY Condition-handling Built-in Function

The ONKEY built-in function returns the key string in the input/output statement for which a TRANSMIT, KEY, RECORD, or ERROR on-unit was invoked.

Format: ONKEY()

Arguments: None

Result Data Type: The data type of the value returned by ONKEY is CHARACTER.

Operation: Use ONKEY() in a KEY, TRANSMIT, RECORD, or ERROR on-unit invoked as the result of an input/output error on a KEYED file. PL/I returns the invalid key value, or the key of the record causing the error.

If you use ONKEY in some other context, or if the on-unit was invoked with a SIGNAL statement, ONKEY returns a null string.

For more information on the ONKEY built-in function, see Chapter 13.

The ONLOC Condition-handling Built-in Function

The ONLOC built-in function returns the name of the entry point for the procedure from which an on-unit was invoked.

Format: ONLOC()

Result Data Type: The data type of the value returned by ONLOC is CHARACTER.

Operation: PL/I returns the name of the entry point of the most recently invoked procedure.

Although you may use ONLOC in any context, it is most useful in an on-unit.

The ONSOURCE Condition-handling Built-in Function

The ONSOURCE built-in function returns the invalid string that caused PL/I to raise the CONVERSION condition.

Format: ONSOURCE()

Arguments: None

Result Data Type: The data type of the value returned by ONSOURCE is CHARACTER.

Operation: Use ONSOURCE() in a CONVERSION or ERROR on-unit invoked as the result of an error in attempting to convert a CHARACTER or pictured-character string to some other data type. PL/I returns the invalid string that gave rise to the conversion error.

If you use ONSOURCE in some other context, or if the CONVERSION or ERROR on-unit is invoked with a SIGNAL statement, PL/I returns a null string.

For more information on the ONSOURCE built-in function, see Chapter 13.

The PAGENO Built-in Function

The PAGENO built-in function returns the page number of the specified PRINT file.

Format: PAGENO(f)

Argument: The argument f must be a scalar with the FILE data type.

Result Data Type: The value returned by the PAGENO built-in function has the integer data type FIXED BINARY(15, 0) REAL.

PL/I Reference Guide

Operation: The file specified by the argument f must be open with the STREAM OUTPUT PRINT attributes. PL/I returns the current output page number.

For more information on page numbers in PRINT files, see Chapter 11.

The POINTER Storage-handling Built-in Function

The POINTER built-in function converts an OFFSET value within a specified area to a POINTER value.

Format: POINTER(o, a)

Arguments: If the argument o is not a scalar, PL/I applies the general rule for aggregate arguments and computes POINTER for each aggregate element. Below, assume that the argument o is a scalar.

The argument o must have the OFFSET data type. The argument a must be a scalar with the AREA data type.

Result Data Type: The data type of the value returned by POINTER is POINTER.

Operation: PL/I interprets the argument o as an offset into the area a and converts the offset to a POINTER value, which PL/I returns.

The PRECISION Arithmetic Built-in Function

The PRECISION built-in function changes the precision of the argument to the specified value.

Format: PRECISION(x, p) or PRECISION(x, p, q)

Abbreviation: PREC for PRECISION

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes PRECISION for each aggregate element. Below, assume that x is a scalar.

For more information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The data type of the value returned by PRECISION has the derived mode, base, and scale of the data type of the argument x. The precision of the data type of the value returned by PRECISION is as described in the section Arguments That Specify Precision.

Operation: PL/I converts the argument x to the result data type and returns that value.

Examples: The following chart illustrates the PRECISION built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	PRECISION(5,4)	5
2	PRECISION(5,4,1)	5.0
3	PRECISION(5E0,4)	5.000E+00
4	PRECISION(5E0,4,1)	ERROR

Lines 1 and 2 illustrate the use of PRECISION with a FIXED argument, and line 3 illustrates PRECISION with a FLOAT argument. The reference in line number 4 is illegal, because the argument q may not be specified for a FLOAT first argument.

The PROD Array-handling Built-in Function

The PROD built-in function returns the product of all the elements of the specified array.

Format: PROD(x)

Argument: The argument x must be an array. It may not be an array of structures.

Result Data Type: The value returned by PROD is a scalar. The data type of the value returned by PROD has the derived base and mode of the data type of an element of the array x. The scale and precision of the data type of the value returned by PROD are determined as follows:

- If the derived scale of the data type of an element of the array x is FIXED and the scale factor of the converted precision is 0, then the data type of the value returned by PROD has a scale of FIXED with a precision of (n,0), where the scale factor is 0, and where n is the maximum number of digits permitted for a data type with a scale of FIXED and the derived base. (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

- Otherwise, the data type of the value returned by PROD has a scale of FLOAT and a precision equal to the converted precision of the data type of an element of the array x.

Operation: PL/I converts each element of the array x to the result data type, and then multiplies all these values together. PL/I returns the product.

Example: Suppose an array A is declared as follows:

```
DECLARE A(4) FLOAT DECIMAL(5) INITIAL(1,2,3,4);
```

Then a reference to PROD(A) returns the product of the elements in the array A and yields +2.4000E1.

The RANK String-handling Built-in Function (Prime Extension)

The RANK built-in function converts a character to its numeric equivalent, according to its position in the ASCII collating sequence (Appendix B).

WARNING

RANK is not an ANS PL/I function and is not available in other implementations of PL/I.

Format: RANK(c)

Arguments: If c is not a scalar, PL/I applies the general rule for aggregate arguments and computes RANK for each aggregate element. Below, assume that c is a scalar.

Note: c must be CHARACTER(1).

Result Data Type: The value returned by RANK has the integer data type FIXED BINARY(15, 0) REAL.

Operation: PL/I takes the bits in the character c and treats them as an unsigned binary integer, which it returns as the function value.

RANK(c) is equivalent to

$\text{INDEX}(\text{COLLATE}(), c) - 1$

Examples: RANK('A') returns 193. RANK('5') returns 181.

The REAL Mathematical Built-in Function

The REAL built-in function returns the real part of a COMPLEX argument.

Format: REAL(x)

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes REAL for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with a mode of COMPLEX, to the derived base and scale of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by REAL has a mode of REAL and the base, scale, and precision of the data type of x.

Operation: PL/I returns the real part of x.

Example: REAL(2 + 3I) returns the value 2.

The REVERSE String-handling Built-in Function

The REVERSE built-in function reverses the order of the characters or bits in the argument and returns the resulting string.

Format: REVERSE(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes REVERSE for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived string type (CHARACTER or BIT) of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by REVERSE is the same as the data type of x.

Operation: PL/I forms a new string containing all the characters or bits in x in reverse order, and returns that string.

Examples: REVERSE('ABCD') returns 'DCBA'. REVERSE('10110'B) returns '01101'B.

The ROUND Arithmetic Built-in Function

The ROUND built-in function rounds a FIXED argument to a specified digit position, and a FLOAT argument to a specified number of significant digits.

Format: ROUND(x, n)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes ROUND for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The argument n must be a decimal integer constant. If the scale of the data type of x is FLOAT, the value of the integer n must be positive.

Result Data Type: The data type of the value returned by ROUND has the same base, scale, and mode as the data type of x. The precision of the value returned by ROUND is determined as follows:

- If the scale of the data type of x is FLOAT, the precision of the result returned by ROUND is n. (If n is larger than the maximum number of digits permitted for a scale of FLOAT with the base of the data type of x, PL/I uses the maximum number of digits permitted.)

- If the scale of the data type of \underline{x} is FIXED, and the precision of the data type of \underline{x} is (r, s) , the precision of the value returned by ROUND is (p, q) , where

$$p = \text{MAX}(1, \text{MIN}(r - s + 1 + n, N))$$

$$q = n$$

where N is the maximum number of digits permitted for a data type with a scale of FIXED and the base of \underline{x} . (N equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I computes a value as follows:

1. If the scale of the data type of \underline{x} is FIXED, PL/I computes a value by rounding \underline{x} in the n th position following the decimal point (or binary point). If n is negative, rounding occurs to the left of the decimal or binary point.
2. If the scale of the data type of \underline{x} is FLOAT, PL/I computes a new value by rounding \underline{x} in the n th significant digit position.

PL/I returns the computed value as the value of ROUND.

Discussion: ROUND does not do what many programmers expect when the first argument is FLOAT. It is tempting to think that ROUND complements the functions CEIL, FLOOR, and TRUNC, by providing a way to round a noninteger value to the nearest integer. In fact, the reference

ROUND(x , 0)

rounds a FIXED value \underline{x} to the nearest integer, but if \underline{x} is FLOAT, the reference is illegal.

The easiest way to round a FLOAT value \underline{x} to the nearest integer is to use FLOOR($x + .5$).

Examples: The following chart illustrates the ROUND built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	ROUND(32.8743,3)	+032.874
2	ROUND(32.8743,1)	+032.9
3	ROUND(32.8743,0)	+033
4	ROUND(32.8743,-1)	+03F1
5	ROUND(32.8743E0,3)	+3.28E1

Line number 1 shows that ROUND(32.8743,3) rounds in the third position after the decimal point, to get the value 32.874, shown as 032.874 in the chart, since the result data type is FIXED DECIMAL(6,4).

Line number 2 shows rounding one position after the decimal point, and line number 3 shows rounding to the nearest integer.

Line number 4 shows rounding to one digit position to the left of the decimal point. ROUND(32.8743,-1) returns the value 30, shown as +03F1 in the chart, since the result data type is FIXED DECIMAL(2,-1).

Line number 5 shows rounding to the third significant digit for a FLOAT value.

The SIGN Arithmetic Built-in Function

The SIGN built-in function returns a value +1, 0, or -1, depending upon whether the argument is positive, zero, or negative, respectively.

Format: SIGN(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes SIGN for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The value returned by SIGN has the integer data type FIXED BINARY(15,0) REAL.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Operation: If x is negative, PL/I returns -1. If x is 0, PL/I returns 0. If x is positive, PL/I returns +1.

Examples: The following chart illustrates the SIGN built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	SIGN(23.8)	+1
2	SIGN(0)	0
3	SIGN(-15.4)	-1

The chart illustrates the three cases of SIGN where the arguments are positive, zero, and negative, respectively.

The SIN Mathematical Built-in Function

The SIN built-in function returns the sine of the argument, with the argument measured in radians.

Format: SIN(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes SIN for each aggregate element. Below, assume that x is scalar.

PL/I converts x to a data type with the scale of FLOAT, to the derived base and mode of the data type of x , and to the converted precision of the data type of x . In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by SIN is the same as the data type of the argument x .

Operation: PL/I returns the sine of x .

Examples: SIN(0) returns the value 0E0. SIN(PI/2) returns the value 1.0000E0.

The SIND Mathematical Built-in Function

The SIND built-in function returns the sine of the argument, with the argument measured in degrees.

Format: SIND(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes SIND for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with the scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by SIND is the same as the data type of x.

Operation: PL/I interprets x as an angle measured in degrees and returns the sine of that angle.

Examples: SIND(0) returns the value 0E0. SIND(90.0) returns the value 1.00E0.

The SINH Mathematical Built-in Function

The SINH built-in function returns the hyperbolic sine of the argument.

Format: SINH(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes SINH for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with a scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

Result Data Type: The data type of the value returned by SINH is the same as the data type of x.

Operation: PL/I returns the hyperbolic sine of x.

Examples: SINH(0) returns 0.E+00. SINH(90.0) returns 8.50E+37.

The SIZE Storage-handling Built-in Function (Prime Extension)

The SIZE built-in function returns the size of the storage area occupied by a variable, measured in bits, bytes, words, double words, or quadruple words.

WARNING

SIZE is not an ANS PL/I function and is not available in other implementations of PL/I.

Format: SIZE(v) or SIZE(v, n)

Arguments: The argument v must be an unsubscripted scalar or aggregate variable. That is, it can be an array, but not an element of an array.

The argument n, if specified, must be one of the following integer constants: 1, 2, 3, 4, or 5.

Result Data Type: The data type of the value returned by SIZE is FIXED BINARY(31).

Operation: If n is not specified, let $n = 3$. The value of n must be between 1 and 5.

PL/I computes the size of the storage area occupied by v, where the unit of storage area is determined by the value of n according to the following chart. PL/I returns the computed value.

<u>Value of n</u>	<u>Unit</u>
1	bit
2	byte (8 bits)
3	word (16 bits)
4	double word (32 bits)
5	quadruple word (64 bits)

The SOME String-handling Built-in Function

The **SOME** built-in function tests whether any of the bits in a BIT scalar or aggregate are 1-bits and returns a logical value to indicate the result.

Format: **SOME(x)**

Argument: The argument **x** must be either a scalar with the BIT data type or an aggregate each of whose scalar elements has the BIT data type. (If **x** is an aggregate, it does not follow the general rule for aggregate arguments.)

Result Data Type: The data type of the value returned by **SOME** is BIT. The length of the string returned by **SOME** is 1.

Operation: If **x** (or any scalar element of the aggregate **x**) contains a 1-bit, PL/I returns '1'B; otherwise, PL/I returns '0'B.

Notice that as a consequence of the above, if **x** is a null BIT string, **SOME** returns '0'B.

Examples: **SOME('000100'B)** returns '1'B. **SOME('000000'B)** returns '0'B.

Consider the following statements:

```
DECLARE A(10);  
  . . .  
  IF SOME(A > 0) THEN . . .
```

In this example, **SOME(A > 0)** is considered true if at least one element of the array **A** is positive.

The SQRT Mathematical Built-in Function

The **SQRT** built-in function returns the square root of the argument.

Format: **SQRT(x)**

Arguments: If **x** is not a scalar, PL/I applies the general rule for aggregate arguments and computes **SQRT** for each aggregate element. Below, assume that **x** is a scalar.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

PL/I converts \underline{x} to a data type with a scale of FLOAT, to the derived base and mode of the data type of \underline{x} , and to the converted precision of the data type of \underline{x} . In the following, assume that \underline{x} is the converted value.

Result Data Type: The data type of the value returned by SQRT is the same as the data type of \underline{x} .

Operation: PL/I returns a value as follows:

1. If the mode of the data type of \underline{x} is REAL, the value of \underline{x} must be 0 or positive; if \underline{x} is negative, the reference is illegal. PL/I returns the positive square root of \underline{x} .
2. If the mode of the data type of \underline{x} is COMPLEX, PL/I returns one of the two COMPLEX square roots of \underline{x} , $u + vI$, such that either \underline{u} is greater than 0, or \underline{u} is 0 and \underline{v} is 0 or greater than 0.

Examples: SQRT(4.0000) returns 2.0000E0. SQRT(-4.0000) is an illegal reference, since the argument is REAL and negative. SQRT(-4 + 0I) returns 0 + 2I (rather than 0 - 2I).

The STRING String-handling Built-in Function

The STRING built-in function concatenates together all elements of a string aggregate.

Format: STRING(s)

Argument: The argument \underline{s} may be a scalar or an aggregate. If it is an aggregate, it does not follow the general rule for aggregate arguments.

Result Data Type: The value returned by STRING is a scalar. The data type of the result returned by STRING is the common derived string type (CHARACTER or BIT) of all the scalar elements in \underline{s} .

Operation: PL/I converts each of the scalar elements in \underline{s} to the result data type and concatenates all of them together. PL/I returns the concatenated string.

Example: Consider the following declaration:

```
DECLARE 1 REC,  
      2 A CHARACTER(3) INITIAL('ABC'),  
      2 B FIXED DECIMAL(2) INITIAL(12),  
      2 C BIT(10) VARYING INITIAL('101'B);
```

The common derived string type of the three elements of the aggregate REC is CHARACTER, and so STRING(REC) returns 'ABC12101'.

The SUBSTR String-handling Built-in Function

The SUBSTR built-in function returns the specified substring of a given string.

Format: SUBSTR(s, m, n) or SUBSTR(s, m)

If s, m, and n are not all scalars, PL/I applies the general rule for aggregate arguments and computes SUBSTR for each aggregate element. Below, assume that s, m, and n are scalar.

PL/I converts s to the derived string type (CHARACTER or BIT) of the data type of s. In the following, assume s is the converted value.

PL/I converts m and n (if specified) to the integer data type FIXED BINARY(15, 0) REAL. In the following, assume that m and n (if specified) are the converted values.

Result Data Type: The data type of the value returned by SUBSTR is the same as the data type of s.

Operation: Let $k = \text{LENGTH}(s)$.

If n is not specified, let

$$n = k - m + 1$$

PL/I tests the following inequalities, if STRINGRANGE checking is enabled:

$$\begin{aligned} 1 &\leq m \leq k + 1 - n \\ 0 &< n \end{aligned}$$

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

If these inequalities are not both satisfied, PL/I raises the STRINGRANGE condition. However, if STRINGRANGE is disabled and these inequalities are not both satisfied, the result is undefined.

PL/I returns a substring of s starting at character or bit position m and continuing for n characters or bits.

Examples: The following chart illustrates the SUBSTR built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	SUBSTR('ABCDEFGH',5,2)	'EF'
2	SUBSTR('ABCDEFGH',5,0)	STRINGRANGE
3	SUBSTR('ABCDEFGH',3)	'CDEFGH'
4	SUBSTR('101101'B,3,2)	'11'B

Line number 1 computes the substring of 'ABCDEFGH', starting at the fifth character and going for two characters, and so returns 'EF'.

In line 2, the starting position is the same, but the length is 0, and so PL/I returns the null string.

Line 3 illustrates the fact that when the third argument is unspecified, the substring goes to the end of the string.

Line 4 illustrates SUBSTR with a BIT string argument.

The SUBTRACT Arithmetic Built-in Function

The SUBTRACT built-in function returns the difference of two values in the specified precision.

Format: SUBTRACT(x, y, p) or SUBTRACT(x, y, p, q)

Arguments: If x and y are not both scalar, PL/I applies the general rule for aggregate arguments and computes SUBTRACT for each aggregate element. Below, assume that x and y are scalar.

PL/I converts x and y to the common derived base, scale, and mode of the data types of x and y, and to the converted precision of the respective argument. In the following, assume that x and y are the converted values.

For information on the arguments p and q, see the section Arguments That Specify Precision near the beginning of this chapter.

Result Data Type: The base, scale, and mode of the data type of the value returned by SUBTRACT are the same as the common derived base, scale, and mode of the data types of x and y. The precision of the data type of the value returned by SUBTRACT is as described in the section Arguments That Specify Precision near the beginning of this chapter.

Operation: PL/I computes the value of $x - y$, and converts it to the result data type.

Examples: The following chart illustrates the SUBTRACT built-in function:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	SUBTRACT(3,5,1)	-2
2	SUBTRACT(3,5,3)	-2
3	SUBTRACT(3,5,3,1)	-2.0
4	SUBTRACT(5.2,3E0,4)	2.200E0
5	SUBTRACT(5.2,3E0,4,1)	ERROR

Lines 1 through 3 illustrate the computation of 3 minus 5 with various result precisions. Line 4 illustrates SUBTRACT with a FLOAT argument.

The reference in line 5 is illegal, since the argument q may not be specified with a FLOAT argument.

The SUM Array-handling Built-in Function

The SUM built-in function returns the sum of all the elements of the array x.

Format: SUM(x)

Arguments: The argument x must be an array. It may not be an array of structures.

Result Data Type: The value returned by SUM is a scalar. The base, scale, and mode of the data type of the value returned by SUM are the derived base, scale, and mode of the data type of an element of the array x. The precision of the data type of the value returned by SUM is determined as follows:

- If the scale of the data type of an element of the array x is FIXED, the precision of the data type of the value returned by SUM is (n, q) , where q is the converted scale factor of the data type of an element of the array x, and n is the maximum number of digits for a data type with a scale of FIXED and the derived base. (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)
- If the scale of the data type of an element of the array x is FLOAT, the precision of the data type of the value returned by SUM is the converted precision of the data type of an element of the array x.

Operation: PL/I converts each element of the array x to the result data type and adds all these values together. PL/I returns the sum.

Example: Suppose the array A is declared as follows:

```
DECLARE A(4) FLOAT DECIMAL(5) INITIAL(1,2,3,4);
```

Then SUM(A) returns the sum of the elements of the array A, or 1.000E1.

The TAN Mathematical Built-in Function

The TAN built-in function returns the tangent of an argument, with the argument given in radians.

Format: TAN(x)

Arguments: If x is not scalar, PL/I applies the general rule for aggregate arguments and computes TAN for each aggregate element. Below, assume that x is scalar.

PL/I converts x to the data type with a scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by TAN is the same as the data type of x.

Operation: If the value of x is an odd multiple of $\pi/2$, the reference is illegal.

PL/I returns the tangent of x.

Examples: TAN(0) returns the value 0E0. TAN($\pi/4$) returns the value 1.0000E0.

The TAND Mathematical Built-in Function

The TAND built-in function returns the tangent of an argument, with the argument given in degrees.

Format: TAND(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes TAND for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to a data type with a scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The data type of the value returned by TAND is the same as the data type of x.

Operation: If the value of x is an odd multiple of 90 degrees, the reference is illegal.

PL/I interprets x as an angle measured in degrees and returns the tangent of that angle.

Example: TAND(0) returns the value 0E0. TAND(45.00) returns the value 1.000E0.

The TANH Mathematical Built-in Function

The TANH built-in function returns the hyperbolic tangent of the argument.

Format: TANH(x)

Argument: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes TANH for each aggregate element. Below, assume that x is a scalar.

PL/I converts x to the data type with a scale of FLOAT, to the derived base and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is the converted value.

Result Data Type: The data type of the value returned by TANH is the same as the data type of x.

Operation: PL/I returns the value of the hyperbolic tangent of x.

Examples:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	TANH(0)	0.E+00
2	TANH(PI4)	1.000000000E+00
3	TANH(45.00)	1.000E+00

The TIME Built-in Function

The TIME built-in function returns the current time of day as a CHARACTER string, in the format 'hhmmssfff'.

Format: TIME()

Arguments: None

Result Data Type: The data type of the value returned by TIME is CHARACTER. The length of the string returned by CHARACTER is 9.

Operation: PL/I returns a CHARACTER(9) value in the format 'hhmmssfff', where hh represents the hours from 00 to 23, mm represents the minutes from 00 to 59, ss represents the number of seconds from 00 to 59, and fff is the fraction of a second in milliseconds from 000 to 999.

Example: At precisely 1:23:45 p.m., TIME() returns the value '132345000'.

The TRANSLATE String-handling Built-in Function

The TRANSLATE built-in function translates a string argument by replacing characters in the string on a one-to-one basis with other characters.

Format: TRANSLATE(s, r, t) or TRANSLATE(s, r)

Arguments: If s, r, and t (if specified) are not all scalar, PL/I applies the general rule for aggregate arguments and computes TRANSLATE for each aggregate element. Below, assume that s, r, and t are all scalars.

PL/I converts each of the arguments s, r, and t (if specified) to CHARACTER. In the following, assume that s, r, and t are the converted values.

Result Data Type: The data type of the value returned by TRANSLATE is CHARACTER. The length of the string returned by TRANSLATE is the same as the length of the string s.

Operation: If t is not specified, let t be a CHARACTER(256) string containing the entire collating sequence. (This is the value returned by a reference to the COLLATE built-in function.)

If string r is shorter than string t, PL/I pads r with blanks to the length of t.

PL/I takes the CHARACTER string s and creates a new CHARACTER string v, by taking s and making one-to-one character substitutions to get v. PL/I does this as follows:

For each character in s, PL/I searches for that character in t. If the character is not in string t, PL/I uses that character untranslated in string v. Otherwise, suppose the leftmost occurrence of the character is at position n in string t. Then PL/I translates that character to the character appearing in position n in string r.

BUILT-IN FUNCTIONS AND PSEUDOVARIABLES

PL/I returns the string v.

Since the above description is somewhat difficult to understand, the following example provides a user-defined function procedure that performs the same function as TRANSLATE.

```
TRANSLATE: PROC(S, R, T) RETURNS(CHAR(*));
DCL (S, R, T) CHAR(*);
DCL V CHAR(LENGTH(S));
DCL C CHAR(1);
DCL (K, M) BIN FIXED;
V = S;
DO M = 1 TO LENGTH(V);
  C = SUBSTR(V, M, 1);
  K = INDEX(T, C);
  IF K > 0 THEN DO;
    IF K > LENGTH(R)
      THEN C = '';
    ELSE C = SUBSTR(R, K, 1);
    SUBSTR(V, M, 1) = C;
  END;
END;
END TRANSLATE;
```

Discussion and Examples: Use the TRANSLATE built-in function to take a CHARACTER string and translate the characters on a one-to-one basis. The translation lists are specified by the second and third arguments.

As an example, consider the following reference:

```
TRANSLATE('AMICABLY', 'XYZ', 'ABC')
```

The second and third arguments to TRANSLATE set up a "translate" table, as shown below.

A	B	C
↓	↓	↓
X	Y	Z

PL/I Reference Guide

When this translate table is applied to the string 'AMICABLY', following substitutions are obtained:

A -> X	Substitute X for A
M -> M	No substitution
I -> I	No substitution
C -> Z	Substitute Z for C
A -> X	Substitute X for A
B -> Y	Substitute Y for B
L -> L	No substitution
Y -> Y	No substitution

Therefore, the above reference to TRANSLATE returns 'XMIZXYLY'.

Similarly, consider the reference:

```
TRANSLATE('BROAD', 'XXZ', 'ABC')
```

This reference uses the same translation table as the preceding example, because the second and third arguments are the same. When this translation table is applied to the string 'BROAD', the following character translations are obtained:

B -> X	Substitute X for B
R -> R	No substitutions
O -> O	No substitutions
A -> X	Substitute X for A
D -> D	No substitutions

Therefore, the above reference would return 'XROXD'.

One of the most common uses of the TRANSLATE built-in function is to change all lowercase letters in a string to uppercase characters. Consider the following program segment:

```
DCL UPPER CHAR(26) INIT('ABCDEFGHIJKLMNOPQRSTUVWXYZ');
DCL LOWER CHAR(26) INIT('abcdefghijklmnopqrstuvwxyz');
. . .
GET LIST(STR);
STR = TRANSLATE(STR, UPPER, LOWER);
```

The last statement changes all lowercase letters in the string STR to their uppercase equivalents.

If you omit the third argument after TRANSLATE, PL/I uses a default argument that is a string containing the entire collating sequence. This form is used in systems programming applications where CHARACTER strings are being translated from one collating sequence to another.

The TRIM String-Handling Function

The TRIM function removes occurrences of a character from a string.

Format: TRIM(s, b) or TRIM(s, b, c)

Arguments: If s and c (if specified) are not scalar, PL/I applies the general rule for aggregate arguments and computes TRIM for each aggregate element. Below, assume that s and c are both scalars.

PL/I converts the value of the argument s to the data type CHARACTER. In the following, assume that s is the converted value.

The argument b must be scalar. PL/I converts the value of the argument b to the data type BIT(2). In the following, assume that b is the converted value.

PL/I converts the value of the argument c, if specified, to the data type CHAR(1). In the following, assume that c is the converted value.

Result Data Type: The data type of the value returned by TRIM is CHARACTER.

Operation: If c is not specified, let c = ' '.

PL/I takes the string s and creates a new CHARACTER string v by starting with v = s and proceeding as follows:

- If the first bit of b is a 1-bit, PL/I removes any initial occurrences of the character c from v.
- If the second bit of b is a 1-bit, PL/I removes any trailing occurrences of the character c from v. PL/I returns the string v.

Examples:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	TRIM('XXABCXX', '00'B, 'X')	'XXABCXX'
2	TRIM('XXABCXX', '01'B, 'X')	'XXABC'
3	TRIM('XXABCXX', '10'B, 'X')	'ABCXX'
4	TRIM('XXABCXX', '11'B, 'X')	'ABC'
5	TRIM(' ABC ', '11'B)	'ABC'

Lines 1-4 illustrate the removal of the character 'X'. The four possible values in the BIT(2) second argument are illustrated.

Line 5 is similar to line 4 except that the third argument is omitted, and so the space character is removed.

The TRUNC Arithmetic Built-in Function

The TRUNC built-in function truncates the argument to an integer value by throwing away the fractional part.

Format: TRUNC(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes TRUNC for each aggregate element. Below, assume x is a scalar.

PL/I converts x to the derived base, scale, and mode of the data type of x, and to the converted precision of the data type of x. In the following, assume that x is a converted value.

The mode of the data type of x must be REAL; if x is COMPLEX, the reference is illegal.

Result Data Type: The base, scale, and mode of the data type of the value returned by TRUNC are the same as for the data type of x. The precision of the data type of the value returned by TRUNC is determined as follows:

- If the scale of the data type of x is FLOAT, the precision of the data type of the value returned by TRUNC is the same as for the data type of x.

- If the scale of the data type of \underline{x} is FIXED, and the precision of the data type of \underline{x} is (r, s) , then the precision of the data type of the value returned by TRUNC is $(p, 0)$, with a zero scale factor, where

$$p = \text{MIN}(n, \text{MAX}(r - s + 1, 1))$$

and n is the maximum number of digits permitted for a data type with the scale of FIXED and the base of the data type of \underline{x} . (n equals 31 for FIXED BINARY, and 14 for FIXED DECIMAL.)

Operation: PL/I returns an integer value, computed as follows:

1. If the value of \underline{x} is nonnegative, PL/I returns the largest integer value that is less than or equal to \underline{x} .
2. If \underline{x} is negative, PL/I returns the smallest integer value that is greater than or equal to \underline{x} .

Discussion: There are three related functions, CEIL, FLOOR, and TRUNC, each of which changes a REAL value to an integer value. All three functions leave an integer value unchanged. If the argument is not an integer,

- FLOOR returns the next lower integer.
- CEIL returns the next higher integer.
- TRUNC returns the next integer in the direction of 0.

TRUNC is the same as FLOOR for positive arguments, and TRUNC is the same as CEIL for negative arguments.

Note that the ROUND built-in function does not always return an integer value, and so is usually not useful. The easiest way to round an argument \underline{x} to the nearest integer is to use $\text{FLOOR}(\underline{x} + .5)$.

Examples: The following chart illustrates the CEIL, FLOOR, and TRUNC built-in functions:

<u>Line #</u>	<u>Reference</u>	<u>Returns</u>
1	CEIL(2.7)	+03
2	FLOOR(2.7)	+02
3	TRUNC(2.7)	+02
4	CEIL(-2.7)	-02
5	FLOOR(-2.7)	-03
6	TRUNC(-2.7)	-02
7	CEIL(2)	+02
8	FLOOR(2)	+02
9	TRUNC(2)	+02
10	CEIL(12.35E0)	+1.300E1
11	FLOOR(12.35E0)	+1.200E1
12	TRUNC(12.35E0)	+1.200E1

Lines 1 through 3 illustrate CEIL, FLOOR, and TRUNC with a positive argument, 2.7. CEIL(2.7) returns the value 3, which is shown as +03, since the data type of the value returned by CEIL(2.7) is FIXED DECIMAL(2, 0) REAL. FLOOR(2.7) and TRUNC(2.7) both return the value 2.

Lines 4 through 6 illustrate the same functions with negative arguments.

Lines 7 through 9 illustrate the fact that these functions leave the value of an integer argument unchanged.

Lines 10 through 12 illustrate the functions with a FLOAT argument. Notice that the value returned is FLOAT even though it is an integer value.

The UNSPEC Built-in Function

The UNSPEC built-in function returns a BIT string for the internal representation of the argument.

Format: UNSPEC(x)

Argument: The argument x must be a scalar, and must be a reference to a variable.

Result Data Type: The data type of the value returned by UNSPEC is BIT.

Operation: PL/I returns a BIT string for the internal representation of the argument x.

Discussion: Most PL/I functions are designed in such a way that they give the same results under all implementations of PL/I. The intention is that if your PL/I program is run on two different implementations of PL/I, the answers produced will be the same.

Some programmers, especially systems programmers, need to manipulate data in its internal machine format, even though the resulting program does not run correctly on other machines. The UNSPEC built-in function permits a program to get at the internal bit representation of a data item. There is also an UNSPEC pseudovvariable, which allows you to modify the bit representation of any data item.

Examples:

<u>Reference</u>	<u>Returns</u>
UNSPEC(3.1416)	'0000 0000 0000 0011' B
UNSPEC(ABCDE)	'1100 0001 1100 0010 1100 0011 1100 0100 1100 0101' B

The VALID Built-in Function

The VALID built-in function determines whether the CHARACTER string value of a PICTURE variable is valid.

Format: VALID(x)

Arguments: If x is not a scalar, PL/I applies the general rule for aggregate arguments and computes VALID for each aggregate element. Below, assume that x is a scalar.

The argument x must be a variable, and must have the PICTURE attribute. The argument x may be either pictured-numeric or pictured-character.

Result Data Type: The VALID built-in function returns a logical value. The data type of the value returned by VALID is BIT. The length of the string returned by VALID is 1.

Operation: PL/I checks the CHARACTER string value of the PICTURE variable x to see whether it conforms to the picture specification for the data type of x.

If the string conforms, PL/I returns '1'B; otherwise, PL/I returns '0'B.

Example: Consider the following program segment:

```
DECLARE VALS(20) PICTURE '$$$9V.99CR';  
..  
READ FILE(TAPE) INTO(VALS);  
IF ^ EVERY(VALID(VALS))  
    THEN PUT SKIP LIST('INVALID TAPE RECORD');
```

The IF statement prints an error message if any of the CHARACTER string values in the array VALS do not conform to the picture specification.

The VERIFY String-handling Built-in Function

The VERIFY built-in function tests a CHARACTER string to determine whether all the characters in it are legal, in the specified sense.

Format: VERIFY(*s*, *c*)

Arguments: If *s* and *c* are not both scalar, PL/I applies the general rule for aggregate arguments and computes VERIFY for each aggregate element. Below, assume that *s* and *c* are scalar.

PL/I converts *s* and *c* to CHARACTER. In the following, assume that *s* and *c* are the converted values.

Result Data Type: The value returned by VERIFY has the integer data type FIXED BINARY(15,0) REAL.

Operation: PL/I tests each character of *s* to see whether that character also appears in the string *c*. PL/I proceeds as follows:

1. If *s* is a null string, or if each character of the string *s* also occurs in the string *c*, PL/I returns the value 0.
2. Otherwise, let *k* be the position of the leftmost occurrence of a character of *s* that does not also occur in *c*. Then PL/I returns the value *k*.

Discussion and Examples: You can use the VERIFY function to determine whether a CHARACTER string contains only legal characters.

For example, suppose an input string is supposed to contain a signed decimal integer. You may use VERIFY to test whether all the characters are legal, as follows:

```
DCL S CHAR(200) VAR;
. . .
GET LIST(S);
K = VERIFY(S, '+-.0123456789');
IF K > 0 THEN
    PUT LIST('INVALID INPUT CHAR', SUBSTR(S, K, 1));
```

The assignment statement sets the variable K equal to the position of the first character in S that is not a digit, decimal point, or sign. If every character in S is a digit, decimal point, or sign, the assignment statement sets K to 0. If there is an invalid character, the PUT statement prints out the invalid character.

Another common use of VERIFY is to find the first nonblank character in a string. The reference

```
VERIFY(S, ' ')
```

returns 0 if S is null or all blanks, and otherwise returns the position of the first nonblank in S.

THE USE OF PSEUDOVARIABLES

You may use certain built-in function names as pseudovariables by assigning them a value. The functions that may be used in this way are IMAG, ONCHAR, ONSOURCE, PAGENO, REAL, STRING, SUBSTR, and UNSPEC.

If the first argument to the SUBSTR, IMAG, or REAL pseudovariable is not a scalar, an aggregate assignment is performed, according to the rules given in Chapter 6.

The IMAG and REAL Pseudovariables

If x is a numeric variable whose data type has a mode of COMPLEX, you may assign a value to just the real or imaginary part of x by using the REAL or IMAG pseudovariables, respectively. The syntax is either of the following:

```
REAL(x) = expression;
IMAG(x) = expression;
```

For example, consider the following program segment:

```
DECLARE C FLOAT COMPLEX;  
..  
REAL(C) = 0;  
IMAG(C) = 5;
```

The two assignment statements, taken together, are equivalent to the following single assignment statement:

```
C = 0 + 5I;
```

The ONCHAR and ONSOURCE Pseudovariab

The ONCHAR and ONSOURCE pseudovariab

The PAGENO Pseudovariab

Use the PAGENO pseudovariab

```
PAGENO(f) = expression;
```

is legal provided that f is a scalar with the FILE attribute, and that the file has been previously opened with the STREAM OUTPUT PRINT attributes. The assignment statement changes the page number for the PRINT file f to the value of the expression.

The STRING Pseudovariab

The STRING pseudovariab

```
STRING(x) = expression;
```

where x is a string scalar or aggregate. All elements of x must be NONVARYING UNALIGNED strings. One of the following must be true:

- All elements of x are BIT.
- All elements of x are CHARACTER, pictured-character, or pictured-numeric.

This means that mixing BIT and CHARACTER strings is illegal.

For example, consider the following program segment:

```
DECLARE C(5) CHARACTER(1);
..
STRING(C) = 'ABCDE';
```

The string array C in the assignment statement is treated as one single string containing five characters.

The SUBSTR Pseudovariable

Assign a value to the SUBSTR pseudovariable to change the value of a substring of a string. The syntax is

```
SUBSTR(s, m, n) = expression;
```

to change the substring of string s starting from position m and going for n characters or bits. Alternatively, use the syntax

```
SUBSTR(s, m) = expression;
```

to change the substring of string s starting from position m and going to the end of string s.

For example, consider the following program segment:

```
DECLARE CV CHARACTER(200) VARYING;
..
CV = 'ABCDEFGH';
SUBSTR(CV,5,2) = 'XY';
```

After the last assignment statement, CV has the value 'ABCDXYGH', since the assignment statement changes the two characters of CV starting at position 5.

Continuing this example, suppose PL/I executes the statement

```
SUBSTR(CV,4) = 'QRSTU';
```

PL/I changes the substring of C starting from position 4 and going to the end of the string, with the result that CV is given the value 'ABCQRSTU'.

Assignment to the SUBSTR pseudovvariable is always treated as a NONVARYING assignment, even when the first argument is a VARYING variable. Therefore, continuing the above example, the statement

```
SUBSTR(CV,4) = 'QRS';
```

would change the value of CV to 'ABCQRSbb', where b stands for a blank character.

A consequence of this rule is that assignment to a SUBSTR of a VARYING string never sets or changes the lengths of the string. This means that it is illegal to use the SUBSTR pseudovvariable on an undefined VARYING string variable.

The restrictions on the second and third arguments to the SUBSTR pseudovvariable are the same as for the SUBSTR built-in function. If they are out of range, PL/I raises the STRINGRANGE condition, if enabled.

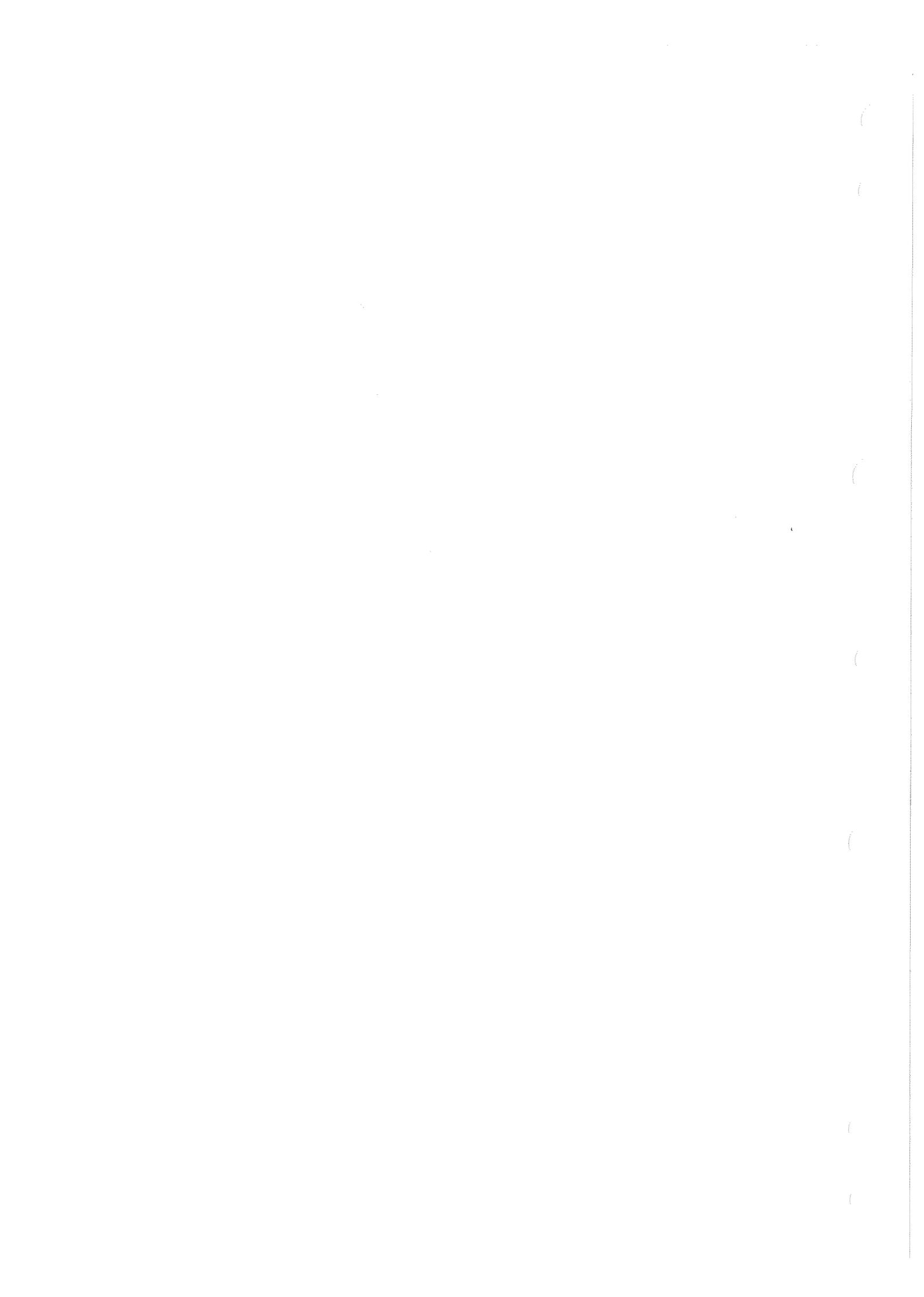
The UNSPEC Pseudovvariable

Just as the UNSPEC built-in function allows you to fetch the value of a data item as a BIT string in the internal computer format, the UNSPEC pseudovvariable allows you to set the value of a data item by specifying the bit configuration that you wish to store. The syntax is

```
UNSPEC(x) = expression;
```

The argument x must be a scalar variable. PL/I converts the expression to BIT and stores the BIT string in the storage area occupied by the variable x.

APPENDIXES



A

PL/I Keywords

LIST OF KEYWORDS

All language elements of Prime PL/I are shown below.
Prime extensions to ANSI Standard PL/I are underlined.
(P) after a keyword means that it is also a pseudovisible.

Statement Keywords and Statement Option Keywords

ALLOCATE	DELETE
IN	FILE
SET	KEY
[assignment statement]	DO
BY NAME	TO
BEGIN	BY
CALL	WHILE
CLOSE	REPEAT
ENVIRONMENT	<u>UNTIL</u>
FILE	END
DECLARE	ENTRY
DEFAULT	OPTIONS (NONQUICK)
ERROR	RECURSIVE
NONE	RETURNS
RANGE	
SYSTEM	

PL/I Reference Guide

FORMAT	PROCEDURE
A	OPTIONS (MAIN)
C	OPTIONS (NONQUICK)
COLUMN	RECURSIVE
E	RETURNS
F	PUT
LINE	DATA
P	EDIT
PAGE	DO
R	FILE
SKIP	LINE
TAB	LIST
X	DO
FREE	PAGE
IN	SKIP
GET	STRING
COPY	TAB
DATA	READ
EDIT	FILE
DO	IGNORE
FILE	INTO
LIST	KEY
DO	KEYTO
SKIP	SET
STRING	RETURN
GOTO (GO TO)	REVERT
IF	REWRITE
THEN	FILE
ELSE	FROM
<u>LEAVE</u>	KEY
LOCATE	<u>SELECT</u>
FILE	<u>WHEN</u>
KEYFROM	<u>OTHERWISE</u>
SET	SIGNAL
ON	STOP
SNAP	WRITE
SYSTEM	FILE
OPEN	FROM
DIRECT	KEYFROM
ENVIRONMENT	<u>%INCLUDE</u>
FILE	<u>%REPLACE</u>
INPUT	<u>%PAGE</u>
KEYED	
LINESIZE	
OUTPUT	
PAGESIZE	
PRINT	
RECORD	
SEQUENTIAL	
STREAM	
TAB	
TITLE	
UPDATE	

Attribute Keywords

ALIGNED	INITIAL
AREA	INPUT
REFER	INTERNAL
AUTOMATIC	KEYED
BASED	LABEL
BINARY	LIKE
BIT	LOCAL
REFER	MEMBER
BUILTIN	NONVARYING
CHARACTER	OFFSET
REFER	OPTIONS
COMPLEX	OUTPUT
CONDITION	PARAMETER
CONSTANT	PICTURE
CONTROLLED	POINTER
DECIMAL	OPTIONS (SHORT)
DEFINED	POSITION
iSUB	PRECISION
DIMENSION	PRINT
REFER	REAL
DIRECT	RECORD
ENTRY	SEQUENTIAL
OPTIONS (SHORTCALL)	STATIC
ENVIRONMENT	STREAM
EXTERNAL	STRUCTURE
FILE	UNALIGNED
FIXED	UPDATE
FLOAT	VARIABLE
FORMAT	VARYING
GENERIC	
WHEN	

Condition Names

AREA	STRINGSIZE
CONDITION	SUBSCRIPTRANGE
CONVERSION	TRANSMIT
ENDFILE	UNDEFINEDFILE
ENDPAGE	UNDERFLOW
ERROR	ZERODIVIDE
FINISH	NOCONVERSION
FIXEDOVERFLOW	NOFIXEDOVERFLOW
KEY	NOOVERFLOW
NAME	NOSIZE
OVERFLOW	NOSTRINGRANGE
RECORD	NOSTRINGSIZE
SIZE	NOSUBSCRIPTRANGE
STORAGE	NOUNDERFLOW
STRINGRANGE	NOZERODIVIDE

PL/I Reference Guide

Arithmetic Built-in Functions

ABS	MAX
ADD	MIN
BINARY	MOD
CEIL	MULTIPLY
DECIMAL	PRECISION
DIVIDE	ROUND
FIXED	SIGN
FLOAT	SUBTRACT
FLOOR	TRUNC

Mathematical Built-in Functions

ACOS	EXP
ASIN	IMAG (P)
ATAN	LOG
ATAND	LOG10
ATANH	LOG2
COMPLEX	REAL (P)
CONJG	SIN
COS	SIND
COSD	SINH
COSH	SQRT
ERF	TAN
ERFC	TAND
	TANH

String-handling Built-in Functions

AFTER	INDEX
BEFORE	LENGTH
BIT	LOW
BOOL	<u>RANK</u>
<u>BYTE</u>	REVERSE
CHARACTER	SOME
COLLATE	STRING (P)
COPY	SUBSTR (P)
DECAT	TRANSLATE
EVERY	TRIM
HIGH	VERIFY

Array-handling Built-in Functions

DIMENSION	LBOUND
DOT	PROD
HBOUND	SUM

Storage-handling Built-in Functions

ADDR	OFFSET
ALLOCATION	POINTER
EMPTY	<u>SIZE</u>
NULL	

Condition-handling Built-in Functions

ONCHAR (P)	ONKEY
ONCODE	ONLOC
ONFIELD	ONSOURCE (P)
ONFILE	

Miscellaneous Built-in Functions

DATE	TIME
LINENO	UNSPEC (P)
PAGENO (P)	VALID



B

The Prime Extended Character Set

As of Rev. 21.0, Prime has expanded its character set. The basic character set remains the same as it was before Rev. 21.0: it is the ANSI ASCII 7-bit set (called ASCII-7), with the 8th bit turned on. However, the 8th bit is now significant; when it is turned off, it signifies a different character. Thus, the size of the character set has doubled, from 128 to 256 characters. This expanded character set is called the Prime Extended Character Set (Prime ECS).

The pre-Rev. 21.0 character set is a proper subset of Prime ECS. These characters have not changed. Software written before Rev. 21.0 will continue to run exactly as it did before. Software written at Rev. 21.0 that does not use the new characters needs no special coding to use the old ones.

Prime ECS support is automatic at Rev. 21.0. You may begin to use characters that have the 8th bit turned off. However, the extra characters are not available on most printers and terminals. Check with your System Administrator to find out whether you can take advantage of the new characters in Prime ECS.

Table B-1 shows the Prime Extended Character Set. The pre-Rev. 21.0 character set consists of the characters with decimal values 128 through 255 (octal values 200 through 377). The characters added at Rev. 21.0 all have decimal values less than 128 (octal values less than 200).

SPECIFYING PRIME ECS CHARACTERS

Direct Entry

On terminals that support Prime ECS, you can enter the printing characters directly; the characters appear on the screen as you type them. For information on how to do this, see the appropriate manual for your terminal.

A terminal supports Prime ECS if

- It uses ASCII-8 as its internal character set, and
- The TTY8 protocol is configured on your asynchronous line.

If you do not know whether your terminal supports Prime ECS, ask your System Administrator.

On terminals that do not support Prime ECS, you can enter any of the ASCII-7 printing characters (characters with a decimal value of 160 or higher) directly by just typing them.

Octal Notation

If you use the Editor (ED), you can enter any Prime ECS character on any terminal by typing

`^octal-value`

where octal-value is the three-digit octal number given in Table B-1. You must type all three digits, including leading zeroes.

Before you use this method to enter any of the ECS characters that have decimal values between 32 and 127, first specify the following ED command:

`MODE CKPAR`

This command permits ED to print as ^nnn any characters that have a first bit of 0.

Character String Notation

The way in which you specify Prime ECS characters in character strings in programs depends on the character that you wish to specify. You can

THE PRIME EXTENDED CHARACTER SET

specify Prime ECS characters on any terminal by using one of the notations shown below. However, the characters themselves can only appear on a terminal that supports Prime ECS. Terminals that do not support Prime ECS will not display the characters correctly.

The following rules describe how to specify Prime ECS characters in character strings.

1. You can specify printing characters in character strings by enclosing them in single quotation marks ('). For example:

'Quoted string'

You can enter the characters using either direct entry or octal notation as described at the beginning of this section.

2. You can specify any character in Prime ECS that has a mnemonic as follows:

\(mnemonic)

where mnemonic is the Prime mnemonic shown for that character in Table B-1. The parentheses are essential. You can specify the mnemonic with either uppercase or lowercase characters. Some characters have more than one mnemonic; you may use any one of these. In the table, the alternatives are separated by a slash character (/). For example:

'A string'\(FF)'with a form feed in it'

The compiler interprets the above example as a single character string.

3. You can specify certain frequently used non-printing characters as

\abbreviation

where abbreviation is one of the following:

<u>Abbreviation</u>	<u>Meaning</u>
B	Backspace
E	Escape
F	Form feed
L	Line feed
N	New line
R	Carriage return
T	Horizontal tab
V	Vertical tab

For example:

'A string'\F'with a form feed in it'

4. You can specify control characters as

\^character

where ^character is listed under "Graphic" in Table B-1. For example:

'A string'\^L'with a form feed in it'

You may use the commercial at sign (@) in place of the caret (^).

A character specified with a backslash (that is, with notation 2, 3, or 4)

- Must appear outside quotation marks
- Specifies a character string of length 1
- Can be specified by itself, or with one or more additional backslash-notation characters, or juxtaposed with one or more quoted character strings

Spaces between the Prime ECS character specification and the character string are not significant, but there must be no spaces within the character specification itself.

THE PRIME EXTENDED CHARACTER SET

The following program example writes three strings that are specified by Prime ECS syntax:

```
ECS_STRING: PROCEDURE OPTIONS(MAIN);
DECLARE SYSPRINT FILE;
DECLARE STR CHAR(32) VAR;
STR = \ (CR) 'HELLO' \ (CR) 'THERE';
PUT LIST(STR);
PUT LIST(\R'HELLO'\R'THERE');
PUT LIST(\^M 'HELLO' \^M 'THERE');
END ECS_STRING;
```

This program produces the following output:

```
HELLO
THERE
HELLO
THERE
HELLO
THERE
```

SPECIAL MEANINGS OF PRIME ECS CHARACTERS

PRIMOS, or an applications program running on PRIMOS, may interpret some Prime ECS characters in a special way. For example, PRIMOS interprets ^P as a process interrupt. ED, the Editor, interprets the backslash (\) as a logical tab. If you wish to make use of the Prime ECS backslash character in a file you are editing with ED, you must define another character as your logical tab.

For a detailed description of how PRIMOS interprets the following Prime ECS characters, see the discussion in the Prime User's Guide of special terminal keys and special characters:

^ \ " ? ^P ^S ^Q _ and ;.

PL/I PROGRAMMING CONSIDERATIONS

Remember that identifiers and program names may contain only letters, numbers, and the dollar sign and underscore characters (\$ and _). These characters form a subset of the ASCII-7 character set.

Character strings, however, can contain any character in Prime ECS. Such strings can be declared as constants, written, read, or assigned to CHARACTER variables.

You can use notations 2, 3, and 4, described above, alone or in juxtaposition with any quoted string in your program. Thus, you can use these notations in constant declarations, assignment statements, and PUT LIST statements.

You cannot, however, use these notations in identifiers or in terminal or file input to GET LIST statements. Therefore, if your terminal does not support Prime ECS, you can enter as terminal input to a GET LIST statement only those characters with decimal numbers greater than 127 (octal numbers greater than 177).

PRIME EXTENDED CHARACTER SET TABLE

Table B-1 contains all of the Prime ECS characters, arranged in ascending order. This order provides both the collating sequence and the way that comparisons are done for character strings. For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal values. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^character; for example, ^P represents the character produced when you type P while holding the control key down.

Characters with decimal values from 000 to 031 and from 128 to 159 are control characters.

Characters with decimal values from 032 to 127 and from 160 to 255 are graphic characters.

THE PRIME EXTENDED CHARACTER SET

Table B-1
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES1	Reserved for future standardization	0000 0000	000	00	000
	RES2	Reserved for future standardization	0000 0001	001	01	001
	RES3	Reserved for future standardization	0000 0010	002	02	002
	RES4	Reserved for future standardization	0000 0011	003	03	003
	IND	Index	0000 0100	004	04	004
	NEL	Next line	0000 0101	005	05	005
	SSA	Start of selected area	0000 0110	006	06	006
	ESA	End of selected area	0000 0111	007	07	007
	HTS	Horizontal tabulation set	0000 1000	008	08	010
	HTJ	Horizontal tab with justify	0000 1001	009	09	011
	VTS	Vertical tabulation set	0000 1010	010	0A	012
	PLD	Partial line down	0000 1011	011	0B	013
	PLU	Partial line up	0000 1100	012	0C	014
	RI	Reverse index	0000 1101	013	0D	015
	SS2	Single shift 2	0000 1110	014	0E	016
	SS3	Single shift 3	0000 1111	015	0F	017
	DCS	Device control string	0001 0000	016	10	020
	PU1	Private use 1	0001 0001	017	11	021
	PU2	Private use 2	0001 0010	018	12	022
	STS	Set transmission state	0001 0011	019	13	023
	CCH	Cancel character	0001 0100	020	14	024
	MW	Message waiting	0001 0101	021	15	025
	SPA	Start of protected area	0001 0110	022	16	026
	EPA	End of protected area	0001 0111	023	17	027
	RES5	Reserved for future standardization	0001 1000	024	18	030
	RES6	Reserved for future standardization	0001 1001	025	19	031
	RES7	Reserved for future standardization	0001 1010	026	1A	032
	CSI	Control sequence introducer	0001 1011	027	1B	033
	ST	String terminator	0001 1100	028	1C	034
	OSC	Operating system command	0001 1101	029	1D	035
	PM	Privacy message	0001 1110	030	1E	036

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	APC	Application program command	0001 1111	031	1F	037
	NBSP	No-break space	0010 0000	032	20	040
¡	INVE	Inverted exclamation mark	0010 0001	033	21	041
¢	CENT	Cent sign	0010 0010	034	22	042
£	PND	Pound sign	0010 0011	035	23	043
¤	CURR	Currency sign	0010 0100	036	24	044
¥	YEN	Yen sign	0010 0101	037	25	045
¦	BBAR	Broken bar	0010 0110	038	26	046
§	SECT	Section sign	0010 0111	039	27	047
¨	DIA	Diaeresis, umlaut	0010 1000	040	28	050
©	COPY	Copyright sign	0010 1001	041	29	051
ª	FOI	Feminine ordinal indicator	0010 1010	042	2A	052
«	LAQM	Left angle quotation mark	0010 1011	043	2B	053
¬	NOT	Not sign	0010 1100	044	2C	054
	SHY	Soft hyphen	0010 1101	045	2D	055
®	TM	Registered trademark sign	0010 1110	046	2E	056
ˆ	MACN	Macron	0010 1111	047	2F	057
°	DEGR	Degree sign	0011 0000	048	30	060
±	PLMI	Plus/minus sign	0011 0001	049	31	061
²	SPS2	Superscript two	0011 0010	050	32	062
³	SPS3	Superscript three	0011 0011	051	33	063
´	AAC	Acute accent	0011 0100	052	34	064
µ	LCMU	Lowercase Greek letter µ, micro sign	0011 0101	053	35	065
¶	PARA	Paragraph sign, Pilgrow sign	0011 0110	054	36	066
•	MIDD	Middle dot	0011 0111	055	37	067
¸	CED	Cedilla	0011 1000	056	38	070
¹	SPS1	Superscript one	0011 1001	057	39	071
º	MOI	Masculine ordinal indicator	0011 1010	058	3A	072
»	RAQM	Right angle quotation mark	0011 1011	059	3B	073
¼	FR14	Common fraction one-quarter	0011 1100	060	3C	074

THE PRIME EXTENDED CHARACTER SET

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
1/2	FR12	Common fraction one-half	0011 1101	061	3D	075
3/4	FR34	Common fraction three-quarters	0011 1110	062	3E	076
¿	INVQ	Inverted question mark	0011 1111	063	3F	077
À	UCAG	Uppercase A with grave accent	0100 0000	064	40	100
Á	UCAA	Uppercase A with acute accent	0100 0001	065	41	101
Â	UCAC	Uppercase A with circumflex	0100 0010	066	42	102
Ã	UCAT	Uppercase A with tilde	0100 0011	067	43	103
Ä	UCAD	Uppercase A with diaeresis	0100 0100	068	44	104
Å	UCAR	Uppercase A with ring above	0100 0101	069	45	105
Æ	UCAE	Uppercase diphthong Æ	0100 0110	070	46	106
Ç	UCCC	Uppercase C with cedilla	0100 0111	071	47	107
È	UCEG	Uppercase E with grave accent	0100 1000	072	48	110
É	UCEA	Uppercase E with acute accent	0100 1001	073	49	111
Ê	UCEC	Uppercase E with circumflex	0100 1010	074	4A	112
Ë	UCED	Uppercase E with diaeresis	0100 1011	075	4B	113
Ì	UCIG	Uppercase I with grave accent	0100 1100	076	4C	114
Í	UCIA	Uppercase I with acute accent	0100 1101	077	4D	115
Î	UCIC	Uppercase I with circumflex	0100 1110	078	4E	116
Ï	UCID	Uppercase I with diaeresis	0100 1111	079	4F	117
Ð	UETH	Uppercase Icelandic letter <u>Eth</u>	0101 0000	080	50	120
Ñ	UCNT	Uppercase N with tilde	0101 0001	081	51	121
Ò	UCOG	Uppercase O with grave accent	0101 0010	082	52	122
Ó	UCOA	Uppercase O with acute accent	0101 0011	083	53	123

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Ô	UCOC	Uppercase O with circumflex	0101 0100	084	54	124
Õ	UCOT	Uppercase O with tilde	0101 0101	085	55	125
Ö	UCOD	Uppercase O with diaeresis	0101 0110	086	56	126
×	MULT	Multiplication sign used in mathematics	0101 0111	087	57	127
Ø	UCOO	Uppercase O with oblique line	0101 1000	088	58	130
Ù	UCUG	Uppercase U with grave accent	0101 1001	089	59	131
Ú	UCUA	Uppercase U with acute accent	0101 1010	090	5A	132
Û	UCUC	Uppercase U with circumflex	0101 1011	091	5B	133
Ü	UCUD	Uppercase U with diaeresis	0101 1100	092	5C	134
Ý	UCYA	Uppercase Y with acute accent	0101 1101	093	5D	135
Þ	UTHN	Uppercase Icelandic letter <u>Thorn</u>	0101 1110	094	5E	136
ß	LGSS	Lowercase German letter double <u>s</u>	0101 1111	095	5F	137
à	LCAG	Lowercase a with grave accent	0110 0000	096	60	140
á	LCAA	Lowercase a with acute accent	0110 0001	097	61	141
â	LCAC	Lowercase a with circumflex	0110 0010	098	62	142
ã	LCAT	Lowercase a with tilde	0110 0011	099	63	143
ä	LCAD	Lowercase a with diaeresis	0110 0100	100	64	144
å	LCAR	Lowercase a with ring above	0110 0101	101	65	145
æ	LCAE	Lowercase diphthong <u>æ</u>	0110 0110	102	66	146
ç	LCCC	Lowercase c with cedilla	0110 0111	103	67	147
è	LCEG	Lowercase e with grave accent	0110 1000	104	68	150
é	LCEA	Lowercase e with acute accent	0110 1001	105	69	151
ê	LCEC	Lowercase e with circumflex	0110 1010	106	6A	152

THE PRIME EXTENDED CHARACTER SET

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ë	LCED	Lowercase e with diaeresis	0110 1011	107	6B	153
ì	LCIG	Lowercase i with grave accent	0110 1100	108	6C	154
í	LCIA	Lowercase i with acute accent	0110 1101	109	6D	155
î	LCIC	Lowercase i with circumflex	0110 1110	110	6E	156
ï	LCID	Lowercase i with diaeresis	0110 1111	111	6F	157
ð	LETH	Lowercase Icelandic letter <u>Eth</u>	0111 0000	112	70	160
ñ	LCNT	Lowercase n with tilde	0111 0001	113	71	161
ò	LCOG	Lowercase o with grave accent	0111 0010	114	72	162
ó	LCOA	Lowercase o with acute accent	0111 0011	115	73	163
ô	LCOC	Lowercase o with circumflex	0111 0100	116	74	164
õ	LCOT	Lowercase o with tilde	0111 0101	117	75	165
ö	LCOD	Lowercase o with diaeresis	0111 0110	118	76	166
÷	DIV	Division sign used in mathematics	0111 0111	119	77	167
ø	LCOO	Lowercase o with oblique line	0111 1000	120	78	170
ù	LCUG	Lowercase u with grave accent	0111 1001	121	79	171
ú	LCUA	Lowercase u with acute accent	0111 1010	122	7A	172
û	LCUC	Lowercase u with circumflex	0111 1011	123	7B	173
ü	LCUD	Lowercase u with diaeresis	0111 1100	124	7C	174
ý	LCYA	Lowercase y with acute accent	0111 1101	125	7D	175
þ	LTHN	Lowercase Icelandic letter <u>Thorn</u>	0111 1110	126	7E	176
ÿ	LCYD	Lowercase y with diaeresis	0111 1111	127	7F	177

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	1000 0000	128	80	200
^A	SOH/TC1	Start of heading	1000 0001	129	81	201
^B	STX/TC2	Start of text	1000 0010	130	82	202
^C	ETX/TC3	End of text	1000 0011	131	83	203
^D	EOT/TC4	End of transmission	1000 0100	132	84	204
^E	ENQ/TC5	Enquiry	1000 0101	133	85	205
^F	ACK/TC6	Acknowledge	1000 0110	134	86	206
^G	BEL	Bell	1000 0111	135	87	207
^H	BS/FE0	Backspace	1000 1000	136	88	210
^I	HT/FE1	Horizontal tab	1000 1001	137	89	211
^J	LF/NL/FE2	Line feed	1000 1010	138	8A	212
^K	VT/FE3	Vertical tab	1000 1011	139	8B	213
^L	FF/FE4	Form feed	1000 1100	140	8C	214
^M	CR/FE5	Carriage return	1000 1101	141	8D	215
^N	SO/LS1	Shift out	1000 1110	142	8E	216
^O	SI/LS0	Shift in	1000 1111	143	8F	217
^P	DLE/TC7	Data link escape	1001 0000	144	90	220
^Q	DC1/XON	Device control 1	1001 0001	145	91	221
^R	DC2	Device control 2	1001 0010	146	92	222
^S	DC3/XOFF	Device control 3	1001 0011	147	93	223
^T	DC4	Device control 4	1001 0100	148	94	224
^U	NAK/TC8	Negative acknowledge	1001 0101	149	95	225
^V	SYN/TC9	Synchronous idle	1001 0110	150	96	226
^W	ETB/TC10	End of transmission block	1001 0111	151	97	227
^X	CAN	Cancel	1001 1000	152	98	230
^Y	EM	End of medium	1001 1001	153	99	231
^Z	SUB	Substitute	1001 1010	154	9A	232
^[ESC	Escape	1001 1011	155	9B	233
^\	FS/IS4	File separator	1001 1100	156	9C	234
]]	GS/IS3	Group separator	1001 1101	157	9D	235
^^	RS/IS2	Record separator	1001 1110	158	9E	236
^_	US/IS1	Unit separator	1001 1111	159	9F	237
	SP	Space	1010 0000	160	A0	240
!		Exclamation mark	1010 0001	161	A1	241
"		Quotation mark	1010 0010	162	A2	242
#	NUMB	Number sign	1010 0011	163	A3	243
\$	DOLR	Dollar sign	1010 0100	164	A4	244
%		Percent sign	1010 0101	165	A5	245
&		Ampersand	1010 0110	166	A6	246

THE PRIME EXTENDED CHARACTER SET

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	1010 0111	167	A7	247
(Left parenthesis	1010 1000	168	A8	250
)		Right parenthesis	1010 1001	169	A9	251
*		Asterisk	1010 1010	170	AA	252
+		Plus sign	1010 1011	171	AB	253
,		Comma	1010 1100	172	AC	254
-		Minus sign	1010 1101	173	AD	255
.		Period	1010 1110	174	AE	256
/		Slash	1010 1111	175	AF	257
0		Zero	1011 0000	176	B0	260
1		One	1011 0001	177	B1	261
2		Two	1011 0010	178	B2	262
3		Three	1011 0011	179	B3	263
4		Four	1011 0100	180	B4	264
5		Five	1011 0101	181	B5	265
6		Six	1011 0110	182	B6	266
7		Seven	1011 0111	183	B7	267
8		Eight	1011 1000	184	B8	270
9		Nine	1011 1001	185	B9	271
:		Colon	1011 1010	186	BA	272
;		Semicolon	1011 1011	187	BB	273
<		Less than sign	1011 1100	188	BC	274
=		Equal sign	1011 1101	189	BD	275
>		Greater than sign	1011 1110	190	BE	276
?		Question mark	1011 1111	191	BF	277
@	AT	Commercial at sign	1100 0000	192	C0	300
A		Uppercase A	1100 0001	193	C1	301
B		Uppercase B	1100 0010	194	C2	302
C		Uppercase C	1100 0011	195	C3	303
D		Uppercase D	1100 0100	196	C4	304
E		Uppercase E	1100 0101	197	C5	305
F		Uppercase F	1100 0110	198	C6	306
G		Uppercase G	1100 0111	199	C7	307
H		Uppercase H	1100 1000	200	C8	310
I		Uppercase I	1100 1001	201	C9	311
J		Uppercase J	1100 1010	202	CA	312
K		Uppercase K	1100 1011	203	CB	313
L		Uppercase L	1100 1100	204	CC	314
M		Uppercase M	1100 1101	205	CD	315
N		Uppercase N	1100 1110	206	CE	316

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	1100 1111	207	CF	317
P		Uppercase P	1101 0000	208	D0	320
Q		Uppercase Q	1101 0001	209	D1	321
R		Uppercase R	1101 0010	210	D2	322
S		Uppercase S	1101 0011	211	D3	323
T		Uppercase T	1101 0100	212	D4	324
U		Uppercase U	1101 0101	213	D5	325
V		Uppercase V	1101 0110	214	D6	326
W		Uppercase W	1101 0111	215	D7	327
X		Uppercase X	1101 1000	216	D8	330
Y		Uppercase Y	1101 1001	217	D9	331
Z		Uppercase Z	1101 1010	218	DA	332
[LBKT	Left bracket	1101 1011	219	DB	333
\	REVS	Reverse slash, backslash	1101 1100	220	DC	334
]	RBKT	Right bracket	1101 1101	221	DD	335
^	CFLX	Circumflex	1101 1110	222	DE	336
_		Underline, underscore	1101 1111	223	DF	337
`	GRAV	Left single quote, grave accent	1110 0000	224	E0	340
a		Lowercase a	1110 0001	225	E1	341
b		Lowercase b	1110 0010	226	E2	342
c		Lowercase c	1110 0011	227	E3	343
d		Lowercase d	1110 0100	228	E4	344
e		Lowercase e	1110 0101	229	E5	345
f		Lowercase f	1110 0110	230	E6	346
g		Lowercase g	1110 0111	231	E7	347
h		Lowercase h	1110 1000	232	E8	350
i		Lowercase i	1110 1001	233	E9	351
j		Lowercase j	1110 1010	234	EA	352
k		Lowercase k	1110 1011	235	EB	353
l		Lowercase l	1110 1100	236	EC	354
m		Lowercase m	1110 1101	237	ED	355
n		Lowercase n	1110 1110	238	EE	356
o		Lowercase o	1110 1111	239	EF	357
p		Lowercase p	1111 0000	240	F0	360
q		Lowercase q	1111 0001	241	F1	361
r		Lowercase r	1111 0010	242	F2	362
s		Lowercase s	1111 0011	243	F3	363
t		Lowercase t	1111 0100	244	F4	364

THE PRIME EXTENDED CHARACTER SET

Table B-1 (continued)
The Prime Extended Character Set

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	1111 0101	245	F5	365
v		Lowercase v	1111 0110	246	F6	366
w		Lowercase w	1111 0111	247	F7	367
x		Lowercase x	1111 1000	248	F8	370
y		Lowercase y	1111 1001	249	F9	371
z		Lowercase z	1111 1010	250	FA	372
{	LBCE	Left brace	1111 1011	251	FB	373
	VERT	Vertical line	1111 1100	252	FC	374
}	RBCE	Right brace	1111 1101	253	FD	375
~	TIL	Tilde	1111 1110	254	FE	376
	DEL	Delete	1111 1111	255	FF	377

C

Data Formats

OVERVIEW

PL/I supports the following data types:

FIXED BINARY
FIXED DECIMAL
FLOAT BINARY
FLOAT DECIMAL
COMPLEX FIXED BINARY
COMPLEX FIXED DECIMAL
COMPLEX FLOAT BINARY
COMPLEX FLOAT DECIMAL
PICTURE
CHARACTER
CHARACTER VARYING
CHARACTER ALIGNED
BIT
BIT VARYING
BIT ALIGNED
POINTER
LABEL
ENTRY
FILE

These data types and arrays of data types are described in Chapter 5. The following descriptions show how the data is internally represented in storage and give some details about each type of data. In the statistics for each data type, P stands for the precision specified when an item of the type is declared.

FIXED BINARY DATA

A 15- or 31-digit two's-complement binary number

Precision: $1 \leq P \leq 31$

Default Precision: (15, 0)

Alignment: word

Storage Requirements: $1 \leq P \leq 15$ for one word
 $16 \leq P \leq 31$ for two words

Internal Representation:

Precision 1-15 (one word):

Bit 1: sign bit
Bits 2-16: digits

Precision 16-31 (two words):

Bit 1: sign bit
Bits 2-32: digits

FIXED DECIMAL DATA

FIXED DECIMAL data is stored as decimal type 3 packed decimal (one decimal digit per four-bit nybble) with a trailing sign nybble. FIXED DECIMAL data is byte-aligned; therefore, the effective precision is always odd. For example, FIXED DECIMAL (4,2) is represented in storage as FIXED DECIMAL (5,2).

Precision: $1 \leq P \leq 14$

Default Precision: (5, 0)

Alignment: byte

Storage Requirements: $\text{FLOOR}((P + 2)/2)$ bytes

Internal Representation:

Each nybble holds one decimal digit. The last nybble holds an indicator of the sign.

FLOAT BINARY DATAPrecision: $1 \leq P \leq 47$ Default Precision: 23Alignment: wordStorage Requirement: $1 \leq P \leq 23$ for two words
 $24 \leq P \leq 47$ for four wordsInternal Representation:

Precision 1-23 (two words):

Bit 1: sign
Bits 2-24: mantissa
Bits 25-32: excess 128 exponent

Precision 24-47 (four words):

Bit 1: sign
Bits 2-48: mantissa
Bits 49-64: excess 128 exponentFLOAT DECIMAL DATAPrecision: $1 \leq P \leq 14$ Default Precision: 6Alignment: wordStorage Requirement: $1 \leq P \leq 6$ for two words
 $7 \leq P \leq 14$ for four wordsInternal Representation:

Precision 1-6 (two words):

Bit 1: sign
Bits 2-24: mantissa
Bits 25-32: excess 128 exponent

Precision 7-14 (four words):

Bit 1: sign
Bits 2-48: mantissa
Bits 49-64: excess 128 exponent

COMPLEX FIXED BINARY DATA

A 31- or 63-digit two's-complement binary number.

Precision: $1 \leq P \leq 31$ for each part

Default Precision: (15, 0) for each part

Alignment: word

Storage Requirements: two or four words, depending on the precision of each part.

Internal Representation for Both Parts:

Precision 1-31 (two words):

Bit 1: sign bit for real portion
Bits 2-16: digits for real portion
Bit 17: sign bit for imaginary portion
Bits 18-31: digits for imaginary portion

Precision 32-63 (four words):

Bit 1: sign bit for real portion
Bits 2-32: digits for real portion
Bit 33: sign bit imaginary portion
Bits 34-63: digits for imaginary portion

COMPLEX FIXED DECIMAL DATA

The two parts of a COMPLEX DECIMAL number are each stored as type 3 packed decimal (one digit per four-bit nybble) with a trailing sign nybble. They are byte-aligned, so the precision is always odd.

Precision: $1 \leq P \leq 14$ for each part

Default Precision: (5, 0) for each part

Alignment: byte

Storage Requirements: $((P + 2)/2)$ bytes

Internal Representation:

Each four-bit nybble holds one decimal digit, except that the last nybble of each part holds the sign.

COMPLEX FLOAT BINARY DATA

Each part of the complex number is stored according to the rules for FLOAT BINARY data above.

Precision: $1 \leq P \leq 47$ for each part

Default Precision: 23 for each part

Alignment: word

Storage Requirement for Each Part: $1 \leq P \leq 23$ for two words
 $1 \leq P \leq 47$ for four words

Internal Representation:

Precision 1-23 (two words) for each part:

Bit 1: sign
 Bits 2-24: mantissa
 Bits 25-32: excess 128 exponent

Precision 24-47 (four words) for each part:

Bit 1: sign
 Bits 2-48: mantissa
 Bits 49-64: excess 128 exponent

COMPLEX FLOAT DECIMAL DATA

Each part of the complex number is stored according to the rules for FLOAT DECIMAL data above.

Precision: $1 \leq P \leq 14$ for each part

Default Precision: 6 for each part

Alignment: word

Storage Requirement for Each Part: $1 \leq P \leq 6$ for two words
 $1 \leq P \leq 14$ for four words

Internal Representation:

Precision 1-6 (two words) for each part:

Bit 1: sign
Bits 2-24: mantissa
Bits 25-32: excess 128 exponent

Precision 7-14 (four words) for each part:

Bit 1: sign
Bits 2-48: mantissa
Bits 49-64: excess 128 exponent

PICTURE DATA

Values to be assigned to a pictured variable are first converted to a decimal value according to the normal conversion rules. This converted value is then used as input to the XED machine instruction, which fills the variable's storage with character data under the control of an edit subprogram. The edit subprogram is placed in the procedure section by the compiler before any generated code.

Picture data is byte-aligned and requires n bytes of storage, where n is the number of picture characters excluding any V character.

CHARACTER DATA

Default Length: 1

Alignment: The ALIGNED attribute has no effect; character data is always byte-aligned.

Storage Requirement: n bytes, where n is the declared length of the string.

Internal Representation:

One character per byte

CHARACTER VARYING DATA

CHARACTER VARYING data is stored as a 16-bit length-word followed by the string value. Only the number of characters specified by the length-word are valid.

Default Length: 1

Alignment: word

Storage Requirements: $\text{FLOOR}((n + 1)/2) + 1$ words. n is the declared maximum length of the string.

Internal Representation:

Bits 1-16 hold the length of the string. Subsequent bytes hold one character per byte.

BIT DATA

Default Length: 1

Alignment: Bit data begins on any bit by default. ALIGNED bit data is word-aligned.

Storage Requirement: $\text{FLOOR}((n + 15)/16)$ words for ALIGNED data, and n bits for unaligned data, where n is the declared length of the string.

Internal Representation:

Each data bit is stored in one hardware bit.

BIT VARYING DATA

Default Length: 1

Alignment: word

Storage Requirement: $\text{FLOOR}((n + 1)/16) + 1$ words, where n is the declared maximum length of the string.

Internal Representation:

Bits 1-16 hold the length of the string. Subsequent bits hold one data bit each.

POINTER DATA

Alignment: word

Storage Requirement: three words

Internal Representation:

Three words are used:

Bit 1: fault code
Bits 2-3: ring number
Bit 4: data format indicator
Bits 5-16: segment number
Bits 17-32: word number
Bits 33-36: bit offset (if bit 4 is set)
Bits 37-48: reserved

POINTER OPTIONS (SHORT)

Alignment: word

Storage Requirement: two words

Internal Representation:

Two words are used:

Bit 1: fault code
Bit 2-3: ring number
Bit 4: data format indicator
Bit 5-16: segment number
Bit 17-32: word number

LABEL DATA

LABEL values are stored as a pair of two-word items. The first item is created by taking the two-word pointer that addresses the code referenced by the label and by interchanging the words so that the word number portion is first. This interchange is performed so that label values may be passed as arguments to routines that expect a FORTRAN-style alternate return argument. The second item is a pointer referencing the stack frame, which should be current after control is transferred to the label.

Alignment: word

Storage Requirements: four words

Internal Representation:

The first two words are the address of the executable statement:

Bits 1-16: word number
 Bit 17: fault code
 Bits 18-19: ring number
 Bit 20: data format indicator (always 0)
 Bits 21-32: segment number

The second two words are the address of the target stack frame:

Bit 33: fault code
 Bits 34-35: ring number
 Bit 36: data format indicator (always 0)
 Bits 37-48: segment number
 Bits 49-64: word number

ENTRY DATA

ENTRY values are stored as a pair of two-word items. The first item is the address of the entry control block (ECB) of the referenced entry. The second item is the first-level display pointer to be used by the invoked procedure. The second pointer value is ignored by EXTERNAL procedures invoked by the entry variable.

Alignment: word

Storage Requirements: four words

Internal Representation:

The first two words are the ECB address:

Bit 1: fault code
 Bits 2-3: ring number
 Bit 4: data format indicator (always 0)
 Bits 5-16: segment number
 Bits 17-32: word number

The second two words are the display pointer):

Bit 33: fault code
 Bits 34-35: ring number
 Bit 36: data format indicator (always 0)
 Bits 37-48: segment number
 Bits 49-64: word number

PL/I Reference Guide

FILE DATA

At Rev. 19, PL/I SEQUENTIAL files are in standard RDBIN/WRBIN subroutine format, and STREAM files are in standard RDASC/WRASC format. (For a discussion of these subroutines, see the PRIMOS Subroutines Reference Guide.) DIRECT files are supported using the subroutine PRWF\$\$ to position to the appropriate word in the file, which is calculated as:

$$(\text{KEYVALUE} * \text{RECORDLENGTH})$$

A FILE data item contains the address of the file control block of the indicated file.

Internal Representation:

Two words are used:

Bit 1: fault code
Bits 2-3: ring number
Bit 4: data format indicator (always 0)
Bits 5-16: segment number
Bits 17-32: word number

ARRAYS

Default Bounds: 1 to the number defined

Maximum Number of Elements: 32,768 per array

Maximum Number of Dimensions: 8

D

Function Return Conventions and Stack Frame Format

The following discussion is presented for the benefit of systems programmers. It assumes a knowledge of Prime's PMA language and operating system.

LOCATIONS OF RETURNED FUNCTION VALUES

<u>Returns Type</u>	<u>Where Returned</u>	
	<u>V-mode</u>	<u>I-mode</u>
fixed bin(1:15)	A-register	GR2 (H)
fixed bin(16:31)	L-register	GR2
float bin(1:23), float dec(1:6)	FAC	FAC1
float bin(24:47), float dec(7:14)	DFAC	DFAC1
bit(1:16)	A-register	GR2 (H)
file	L-register	GR2
pointer	FAR0	FAR0

For all other data types, the calling procedure sets up FAR0 to point to the location at which the function's value is to be returned. When the function becomes active, it transfers the contents of FAR0 to SB%+40 to SB%+42 of its stack frame.

STACK FRAME FORMAT

Figure D-1 shows a typical stack frame format for a PL/I application. The following notes explain the stack frame format entries.

Notes on Figure D-1

- Bit 5 of FLAGS is set for PL/I procedure stack frames.
- Bit 6 of FLAGS is set if dynamic condition handling is taking place, that is, if a condition prefix exists on a statement other than a block or procedure.
- Bit 7 of FLAGS is set if the current stack frame belongs to a PL/I library routine.
- Bit 8 of FLAGS is set if static condition handling is taking place (that is, if a condition prefix exists on a procedure or block), and it is set as the default if no condition prefixes exist. The default setting takes place because the PL/I compiler must load a word containing the ANS PL/I default enablement status of user-modifiable conditions. The ANS PL/I defaults differ from the PRIMOS system defaults.
- SB%+1 to SB%+9 (all offsets referenced in decimal) comprise the hardware-defined portion of the stack.
- SB%+18 points to the ECB (Entry Control Block) of the owning PL/I block, for both procedure and begin blocks. The ECB of a PL/I procedure block is immediately followed by a char(*)var giving the name of the procedure or entry it represents.
- SB%+28 to SB%+33 is defined in the documentation for the CONDITION mechanism in the PRIMOS Subroutines Reference Guide.
- SB%+40 to SB%+42 is always present, whether the block is a function or not. It is the last item in the stack which is guaranteed to be present.
- Display pointers are used by internal blocks to access automatic data declared in containing blocks; there is one for each level of lookback used in the block. Each display pointer is the stack pointer of the corresponding block. The display pointers, if any, begin at SB%+43 and are the last stack item to have a fixed address. Each internal block is PCL'ed with the stack pointer of its parent block in the L-register or GR2; this is

FUNCTION RETURN CONVENTIONS AND STACK FRAME FORMAT

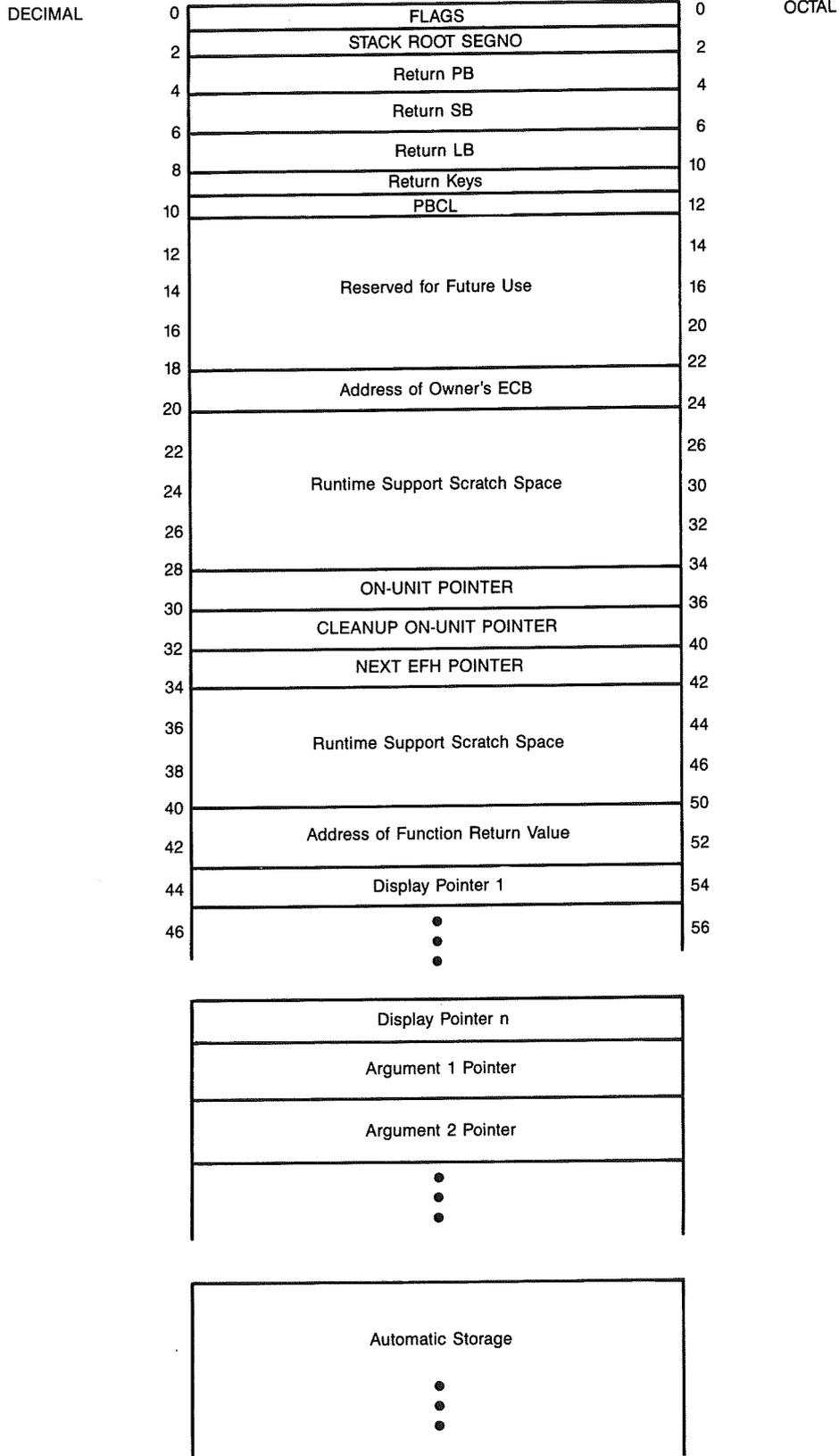


Figure D-1
Stack Frame Format

stored as the first-level display pointer. Additional levels, if needed, are set up by prologue code.

- Argn pointers are the hardware-defined pointers used to reference the parameters of a procedure; they begin immediately following the display pointers.
- Two locations have been designated for use by the PL/I condition-handling mechanism. Stack location SB%+40 and word 9 of the ECB have been reserved for a condition control word.

The condition control word is a 16-bit value whose leftmost nine bits correspond to the enablement status of the nine user-modifiable conditions. If dynamic condition handling is taking place (that is, if bit 6 of FLAGS is set), then location 40 of the user stack contains the control word. If static condition handling is taking place (that is, if bit 8 of FLAGS is set), then word 9 of the ECB contains the control word.

The following chart shows how the bits correspond to the enablement status of the conditions:

<u>Bit</u>	<u>Corresponding Condition</u>	<u>Default Enablement Status</u>
1	CONVERSION	Enabled
2	FIXEDOVERFLOW	Enabled
3	OVERFLOW	Enabled
4	SIZE	Disabled
5	STRINGRANGE	Disabled
6	STRINGSIZE	Enabled
7	SUBSCRIPTRANGE	Disabled
8	UNDERFLOW	Enabled
9	ZERODIVIDE	Enabled
10-16	must be zero	

If the condition is enabled, the bit value is one; if disabled, zero. Thus the condition control word for a PL/I program with no condition prefixes would appear as '1110010110000000'B in ECB word 9. The condition control word for a PL/I program with a dynamic SIZE condition prefix would appear as '1111010110000000'B in stack location 40. For more information about enabling and disabling conditions, see Chapter 13.

E

Differences Among ANS, IBM, and Prime PL/I

PRIME EXTENSIONS TO THE ANSI STANDARD

Prime has implemented a large number of extensions to the ANSI Standard. Customers should not use these extensions if they want to develop PL/I programs which can be compiled on other PL/I implementations. However, if they plan to run their programs exclusively on Prime systems, they can take advantage of powerful mainframe syntax. The following list describes all PL/I extensions at Rev. 19. Extensions also used by IBM are marked with an asterisk.

- READ and WRITE on stream files
- The LEAVE statement
- The SELECT statement
- The BYTE built-in function
- The RANK built-in function
- The SIZE built-in function
- Use of the OPTIONS (SHORT) specifier in DECLARE statements to control space allocation for pointer variables
- Use of the OPTIONS (SHORCALL) specifier in ENTRY declarations to document space allocation for PMA subroutines
- Nonstandard properties of the device named TTY

PL/I Reference Guide

- Use of an A-format without a field width to read a variable-length input line*
- Use of both uppercase and lowercase characters in names
- Acceptance in DO statements of any arithmetic value as an index variable
- Passing of unconnected arrays to asterisked-extent parameters
- Use of OPTIONS (NONQUICK) to specify that a procedure is to be called with the PCL assembly-language instruction
- Overlaying of any data type by another in storage*
- %PAGE, %INCLUDE, and %REPLACE
- %LIST* and %NOLIST* (for IBM %PRINT, %NOPRINT)
- The TITLE options CTLASA, FUNIT, DEVICE, APPEND, NOSIZE, RECL, and FORMS
- The COMPLEX data types*
- Use of the # and \$ characters in names*

ANS FEATURE NOT SUPPORTED IN PRIME PL/I

The use of subscripted label prefixes on PROCEDURE and FORMAT statements is not supported by Prime.

IBM FEATURES NOT SUPPORTED IN PRIME PL/I

Preprocessor Facilities

- The preprocessor facilities of IBM PL/I are not supported. These include the following statements:

%ACTIVATE, %ASSIGN, %DEACTIVATE, %DECLARE, %DO, %END, %GOTO,
%IF, %NOTE, %NULL, %PROCEDURE, %RETURN

- The following listing control statements are not supported:

%CONTROL, %NOPRINT, %PRINT, %SKIP

Note, however, that %NOLIST and %LIST perform the same function as %NOPRINT and %PRINT.

Multitasking Facilities

The multitasking facilities of IBM PL/I are not supported. These include the following features:

- The DELAY, EXIT, and WAIT statements
- The TASK, EVENT, and PRIORITY options of the CALL statement
- The EVENT option of the READ and WRITE statements
- The TASK and EVENT data attributes
- The COMPLETION, PRIORITY, and STATUS built-in functions and pseudovariables

Diagnostic Facilities

The diagnostic facilities of IBM PL/I are not supported. These include the following features:

- The CHECK condition prefix and the CHECK condition
- The CHECK, FLOW, NOCHECK, and NOFLOW statements

Program Elements

- The IBM 48-character set is not supported.
- Blanks are not permitted to appear within compound operators such as <=, >=, ^=, ^<, ||, ->, and **.

Data Elements

Sterling constants are not supported.

Program Structure

- The following options of the PROCEDURE and ENTRY statements are not supported:

ORDER, IRREDUCIBLE, REDUCIBLE, REORDER

- The following options of the OPTIONS option of the PROCEDURE and ENTRY statements are not supported:

COBOL, FORTRAN, NOMAP, NOMAPIN, NOMAPOUT, REENRANT, TASK

Declarations and Attributes

- The following attributes are not supported:

BACKWARDS, BUFFERED, CONNECTED, EVENT, EXCLUSIVE,
IRREDUCIBLE, REDUCIBLE, TASK, TRANSIENT, UNBUFFERED

- The following PICTURE characters are not supported:

G, H, M, P, 6, 7, 8

- The ENVIRONMENT attribute is not supported.
- The arithmetic default attributes are FIXED BINARY.
- The ENTRY declaration for an EXTERNAL procedure may not be omitted, as contextual declaration of an EXTERNAL procedure by its appearance in a CALL statement is not supported.
- The following ENVIRONMENT options are not supported:

F, FB, FS, FBS, V, VB, VBS, D, DB, U, RECSIZE(), BLKSIZE(),
BUFFERS(), BUFND(), BUFNI(), BUFSP(), CONSECUTIVE, INDEXED,
REGIONAL(1), REGIONAL(2), REGIONAL(3), TP(M), TP(R), LEAVE,
REREAD, SIS, SKIP, BKWD, REUSE, TOTAL CILASA, CIL360, COBOL,
INDEXAREA(), NOWRITE, ADDBUFF, GENKEY, NCP(), TRKOFL,
SCALARVARYING, KEYLENGTH(), KEYLOC(), ASCII, BUFOFF(),
PASSWORD()

- The VALUE and DESCRIPTORS options of the DEFAULT statement are not implemented.
- Note that the rules for the promotion of one aggregate type to another differ between IBM PL/I and ANS PL/I.

Built-in Functions

- The following built-in functions are not supported:

ALL, ANY, COMPLETIME, COMPLETION, COUNTER, CURRENTSTORAGE,
DATAFIELD, NULLO, ONCOUNT, PARMSET, PLIRETV, POLY, PRIORITY,
REPEAT, SAMEKEY, STATUS, STORAGE

- The ANSI UNSPEC built-in function requires that the argument be a reference. (In IBM PL/I, the argument may be any expression.)

Program Control

The EXIT statement is not supported.

DIFFERENCES AMONG ANS, IBM, AND PRIME PL/I

Conditions and Exception Control

The following conditions are not supported:

ATTENTION, CHECK, PENDING

Functions and Procedures

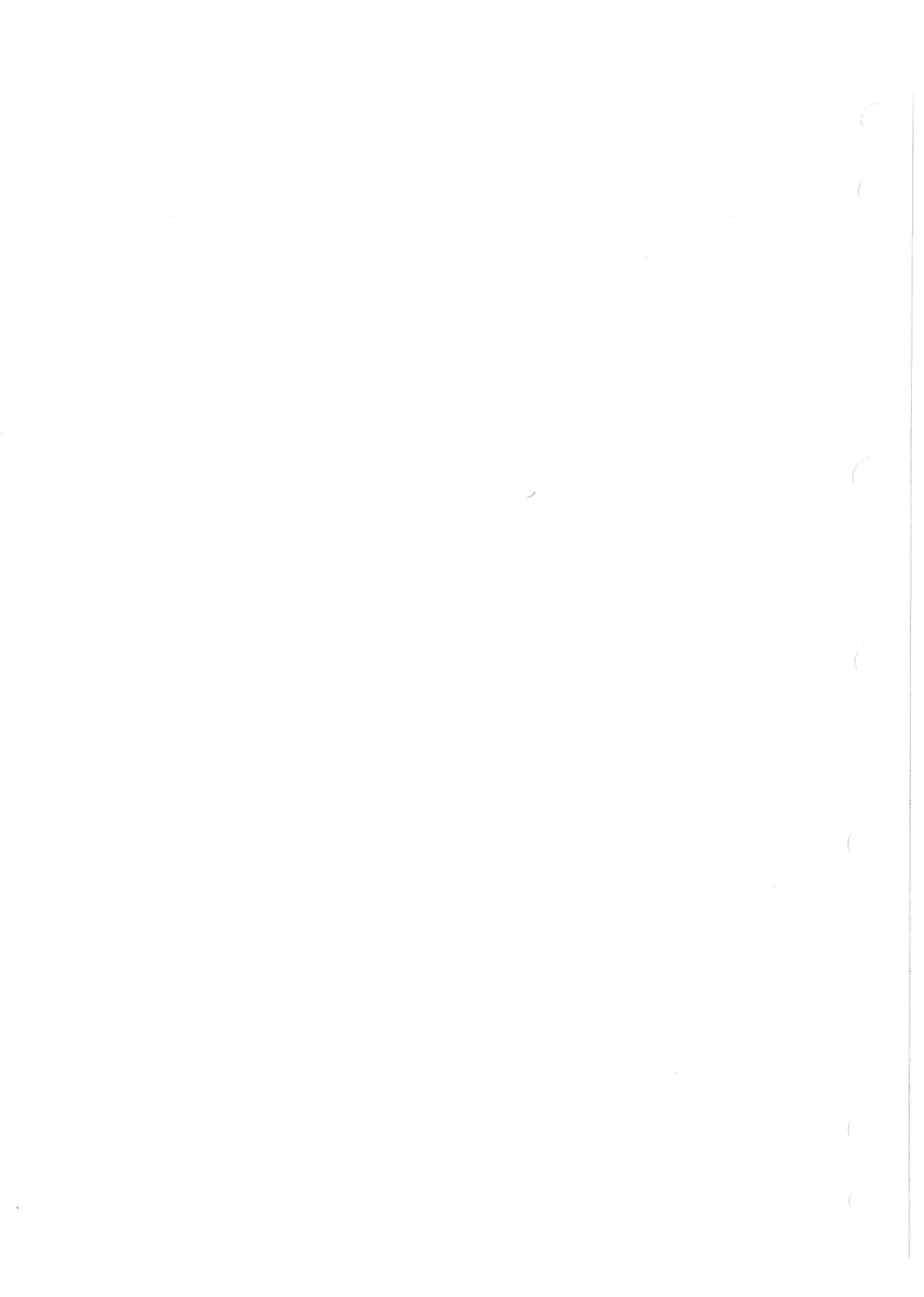
In Prime and ANSI specifications, a reference to a function with no arguments returns an ENTRY value. An empty argument list is required to cause invocation of a function without parameters. (In IBM PL/I, a reference to a function with no arguments causes the implicit invocation of the function if the context does not expect an ENTRY value.)

Input/Output

- The following file attributes are not supported:

BACKWARDS, BUFFERED, EXCLUSIVE, TRANSIENT, UNBUFFERED

- The DISPLAY and UNLOCK statements are not supported.
- The LEAVE, REREAD, and REWIND options of the CLOSE statement are not supported.
- The FLOW and ALL options of the PUT statement are not supported.
- The EVENT option of the READ and WRITE statements is not supported.



F

ONCODE Values

The use of ONCODE in error handling is discussed in Chapters 13 and 14.

1	OPEN of input file for output
2	OPEN of output file for input
3	OPEN with inconsistent attributes
4	OPEN request failed
5	record size too large
6	append to existing file failed
7	OPEN of device already open
8	OPEN of >16 disk files
9	OPEN of SAM file for keyed I/O
10	append to SAM file for keyed I/O
11	input file not found
12	OPEN of device for keyed I/O
13	OPEN of DAM file without DIRECT
14	CLOSE request failed
15	output to input file
16	end of file
17	input from output file
18	input from DIRECT file without key
19	input from keyed file failed
20	input from nonkeyed file with KEY or KEYTO
21	input from nonkeyed file failed
22	output to DIRECT file without key
23	output to nonkeyed file failed
24	output to nonkeyed file with KEYFROM
25	output to nonkeyed file failed
26	DELETE from file not opened as KEYED and UPDATE

PL/I Reference Guide

- 27 DELETE without key
- 28 REWRITE file must be KEYED unless preceded by READ with SET option
- 29 GET from other than STREAM INPUT file
- 30 format nesting or aborted GET/PUT stack too deep
- 31 PUT to other than STREAM OUTPUT file
- 32 invalid char after quoted string in GET LIST
- 33 PAGE, LINE, SKIP(0), or TAB with other than STREAM OUTPUT PRINT file
- 34 field width too small for E format
- 35 field width too small for B format
- 36 field size omitted in B input format
- 37 OPEN with invalid device name
- 38 field size omitted in A input format
- 39 SKIP format used with STRING option
- 40 COLUMN format used with STRING option
- 41 LINE format used with STRING option
- 42 input string exhausted
- 43 input ends inside quoted string
- 44 LOCATE file must be RECORD
- 45 COPY file must be STREAM and OUTPUT
- 46 the disk is full
- 47 READ file with SET option must be RECORD
- 48 REWRITE file must be UPDATE
- 49 REWRITE file must be RECORD
- 50 open with -NOSIZE on non-DIRECT file
- 51 KEY illegally omitted from KEYED SEQUENTIAL READ
- 52 KEYFROM omitted from KEYED SEQUENTIAL WRITE
- 53 KEY illegally omitted from KEYED SEQUENTIAL REWRITE
- 54 FROM illegally omitted from KEYED SEQUENTIAL REWRITE
- 55 KEYFROM omitted from KEYED SEQUENTIAL LOCATE
- 56 KEY illegally omitted from KEYED SEQUENTIAL DELETE
- 57 unable to unlock KEYED SEQUENTIAL record
- 58 FROM illegally omitted from REWRITE
- 59 OPEN of non-MIDAS file as KEYED SEQUENTIAL
- 60 unable to create KEYED SEQUENTIAL file
- 61 OPEN of KEYED SEQUENTIAL file failed
- 62 OPEN of MIDAS file with noncharacter keys
- 63 record size in TITLE option too large
- 64 -CTLASA specified for non-PRINT file
- 1024 ALLOCATE request too large
- 1025 insufficient space for ALLOCATE
- 1026 bad pointer in FREE request
- 1027 decimal FIXEDOVERFLOW in MOD
- 1028 user SIGNAL
- 1029 SQRT argument < 0
- 1030 LOG argument <= 0
- 1031 OVERFLOW in EXP
- 1032 SUBSCRIPTRANGE
- 1033 STRINGRANGE
- 1034 single-precision OVERFLOW/UNDERFLOW
- 1035 double-precision OVERFLOW/UNDERFLOW
- 1036 single-precision floating ZERODIVIDE
- 1037 double-precision floating ZERODIVIDE

1038 single-precision OVERFLOW/UNDERFLOW
 1039 FIXEDOVERFLOW in floating to fixed conversion
 1040 fixed binary ZERODIVIDE/FIXEDOVERFLOW
 1041 decimal FIXEDOVERFLOW
 1042 decimal ZERODIVIDE
 1043 decimal CONVERSION
 1044 no on-unit for ENDFILE
 1045 no on-unit for KEY
 1046 argument range in ASIN/ACOS
 1047 OVERFLOW in TAN
 1048 illegal conversion
 1049 unrecognized data type
 1050 SIZE in conversion
 1051 illegal character in conversion
 1052 OVERFLOW in conversion
 1053 negative argument raised to a floating power
 1054 0 raised to 0 or a negative power
 1055 invalid picture data
 1056 error in picture edit subroutine
 1057 argument range in ATANH
 1058 attempt to evaluate ATAN(0,0) or ATAND(0,0)
 1059 CLOG argument = 0
 1060 complex floating ZERODIVIDE
 1061 reference through null pointer
 1062 improper number of subscripts in GET DATA
 1063 invalid syntax in value field in GET DATA
 1064 in GET DATA
 1065 unmatched "'" in GET DATA
 1066 structure nesting too deep in GET DATA
 1067 name too long in GET DATA
 1068 name not followed by "=" in GET DATA
 1069 invalid syntax in name field in GET DATA
 1070 name not found in GET DATA
 1071 ambiguous reference in GET DATA
 1072 name not in DATA list
 1073 no on-unit for NAME
 1074 no on-unit for CONVERSION
 1075 no on-unit for SUBSCRIPTRANGE
 1076 assignment to BIT NONVARYING
 1077 assignment to CHAR NONVARYING
 1078 assignment to BIT VARYING
 1079 assignment to CHAR VARYING
 1080 FREE request in empty area
 1081 FREE request of already free item
 1082 target area too small for assignment
 1083 no on-unit for AREA
 1084 no on-unit for RECORD
 1085 no on-unit for TRANSMIT
 1086 no on-unit for UNDEFINEDFILE
 1087 illegal character in conversion to BIT
 1088 error in character concatenation
 1089 record size mismatch in READ
 1090 record not found in READ
 1091 unrecoverable READ error

PL/I Reference Guide

1092	record size mismatch in WRITE
1093	record not found in REWRITE
1094	unrecoverable REWRITE error
1095	record size mismatch in WRITE
1096	record already present in WRITE
1097	unrecoverable WRITE error
1098	record not found in DELETE
1099	record not found in LOCATE
1100	illegal key size
1101	single-precision floating point
1102	double-precision floating point
1103	store exception
1104	in float to fixed conversion
1105	fixed binary
1106	fixed decimal
1107	unrecognized arithmetic fault
1108	no on-unit for FIXEDOVERFLOW
1109	no on-unit for OVERFLOW
1110	no on-unit for SIZE
1111	no on-unit for STRINGRANGE
1112	no on-unit for ZERODIVIDE
1113	no on-unit for STORAGE
1114	OPEN of nonexistent INPUT file
1115	OPEN failure
1116	OPEN of non-MIDAS file as KEYED SEQUENTIAL
1117	OPEN of nonexistent KEYED SEQUENTIAL file
1118	OPEN of KEYED SEQUENTIAL file failed
1119	OPEN of MIDAS file with noncharacter keys
1120	error in character picture program
1121	nondigit found in "9" character position
1122	nonalphabetic character found in "A" character position
1123	error in mantissa of floating-point picture program
1124	insufficient precision in target of assignment

G

Glossary of PL/I Terms

activation of BEGIN block

The passing of control to the BEGIN statement in a normal sequential manner. See also BEGIN block.

activation of a PROCEDURE block

The passing of control to a procedure by a procedure reference (a function reference or CALL statement). See also PROCEDURE block.

aggregate

A logical collection of data items. See also scalar, array, structure.

allocation

Association of a specified region of storage with a variable. See also storage class.

argument

An expression in parentheses that is part of a procedure reference. See also parameter, procedure reference.

argument passing

The process by which expressions are transferred from a procedure to a subprocedure. Arguments are passed by reference and by value.

argument passing by reference

Passing an argument in such a way that the parameter and the argument describe the same storage.

argument passing by value

Passing an argument in such a way that argument is copied to a temporary block of storage in the caller's stack frame. This block is then passed to the parameter instead of the actual argument. Any argument that is an expression, a function reference, a built-in function reference, a constant, a parenthesized variable reference, or a reference to a variable with a data type that does not match the parameter, is passed by value.

arithmetic constant

A constant representing a DECIMAL value. (BINARY arithmetic values have no constant representation, but DECIMAL constants can be converted to BINARY by using them in a context that expects a BINARY arithmetic value.) See also constant.

arithmetic data

Values that can be used in computation and whose data type has base, scale, mode, and precision. See also complex data, base, scale, mode, precision, pictured data, string data.

arithmetic operator

A symbol specifying an arithmetic operation. The arithmetic operators are +, denoting addition or prefix positive; -, denoting subtraction or prefix negative; *, denoting multiplication; /, denoting division; and **, denoting exponentiation. See also operator.

array

An n-dimensional ordered set of elements all having the same data type. Elements of an array are referenced by their position within the array. Each array has a specified (declared) number of dimensions, and each dimension has a specified lower and upper bound. See also aggregate, subscript.

array bound

That component of the dimension attribute that defines the upper or lower limit of a dimension of an array.

array of structures

An array whose elements are structures. See also structure.

attribute factoring

Enclosure of a list of identifiers and partial attribute sets in parentheses, followed by the set of attributes that are common to all of them.

automatic storage

Storage that is allocated for a variable when the block in which it is declared is activated and that is released when that block is terminated. See also storage class.

base

The number system used to represent arithmetic data: in PL/I, either BINARY or DECIMAL. See also arithmetic data.

based storage

Controlled storage that is identified and accessed by means of pointer variables. See also controlled storage, pointer, storage class.

BEGIN block

An internal block starting with a BEGIN statement and ending with an END statement. See also block, activation of BEGIN block, termination of BEGIN block.

bit string

A sequence of binary digits that can be operated on with the string functions. See also character string, string data.

bit-string constant

A constant consisting of zero or more binary digits enclosed in single quotation marks followed by the letter B. For example:

'0100'B

Bit-string constants may also be written in quartal, octal and hexadecimal notation. For example:

'231'B2 (quartal notation)

'775'B3 (octal notation)

'A70'B4 (hexadecimal notation)

See also constant.

bit-string operator

A symbol specifying an operation on bit strings. The bit string operators are ^, denoting NOT; &, denoting AND; and | (or !), denoting OR. See also operator.

block

A program section of organized statements beginning with a PROCEDURE or BEGIN statement and ending with the matching END statement. These statements delimit the scope of all identifiers that are declared within the block and that are not given the EXTERNAL attribute. See also activation of BEGIN block, activation of PROCEDURE block, BEGIN block, PROCEDURE block, termination of BEGIN block, termination of PROCEDURE block.

built-in function

A function procedure supplied by the PL/I compiler. See also function, procedure.

character string

A sequence of characters that can be operated on with the string functions. See also bit string, string data.

character-string constant

A constant consisting of zero or more characters enclosed in single quotation marks. For example:

'98.6F'

See also constant.

comparison operator

A symbol specifying a logical operation on bit strings. The comparison operators are $>$, denoting greater than; $\wedge >$, denoting not greater than; \geq , denoting greater than or equal to; $=$, denoting equal to; $\wedge =$, denoting not equal to; \leq , denoting less than or equal to; $<$, denoting less than; and $\wedge <$, denoting not less than. See also operator.

complex data

Arithmetic data consisting of a real part and an imaginary part. See also arithmetic data.

compound statement

A statement that contains another statement. PL/I has two compound statements: the IF statement and the ON statement. See also statement.

condition

An event that takes place during execution of a statement that alters or prevents the normal execution of the statement. See also on-condition, on-unit.

condition name

A keyword that may be specified in ON or SIGNAL statements. The legal condition names are listed in Appendix A.

condition prefix

A parenthesized list of one or more condition names, placed before a statement label, that specifies whether those conditions are enabled. See also scope of a condition prefix.

constant

A sequence of characters that has no name and that represents a particular value that cannot change. Also, an unsubscripted label prefix, a file name, or an entry name. See also arithmetic constant, bit-string constant, character-string constant, file constant, fixed-point constant, floating-point constant, integer constant.

controlled storage

Storage whose allocation and freeing are wholly controlled by ALLOCATE and FREE statements. See also based storage, storage class.

conversion

The change in data type, if any, that is made when a value is assigned to a variable.

data-directed input/output

Transmission of data to or from a stream file in the form of a list of all declared variables and their values. See also edit-directed input/output, list-directed input/output, stream file.

defined variable

A variable that is declared as occupying the same storage as another variable.

direct access

A method of storing records in files that does not require a sequential search of the file to find a particular record. See also key, record file, sequential access.

dynamic storage

Storage that is allocated during execution of a program. See also static storage, storage class.

edit-directed input/output

Transmission of data to or from a stream file by means of a format list. See also format item, format list, data-directed input/output, list-directed input/output, stream file.

entry name

An identifier associated with a procedure, by which reference may be made to the procedure. Procedures that are not part of the compiled module must be declared with the ENTRY attribute and, if they are functions, with the RETURNS attribute. Entry data is used to describe subroutines and functions that are not built-in. (ENTRY is used instead of PROCEDURE because procedures can be entered at points other than the PROCEDURE statement.)

extent

A value that determines a variable's storage size. Extent expressions can be dimension bounds for arrays, maximum string lengths, or AREA sizes. Extents are evaluated when storage is allocated for the variable.

external block

See external procedure.

external procedure

A PROCEDURE block whose entry name is not within the scope of any (other) block. Also called external block. See also procedure.

file

An organized collection of data stored in the computer system's memory. A file is referenced by using a file name declared with the FILE attribute. See also record file, stream file.

file constant

A name declared with the FILE attribute. A file constant cannot be the target of an assignment statement. See also constant.

file control block

A block of STATIC storage associated with each file constant in which information about the current status of the file is kept while the file is open.

file variable

A name declared with the FILE and VARIABLE attributes. A file variable can be assigned file values.

file value

A value designating a file control block that can be opened or closed and that can thereby be connected to various files and devices known to the operating system. File values result from references to file constants, file variables, and file-valued functions.

fixed-point constant

A constant consisting of one or more numeric digits with optional sign and decimal point. For example:

7.5

See also constant.

fixed-point scale

The format of arithmetic data in which the datum is a rational binary or decimal number with a specified number of digits. See also scale.

floating-point constant

A constant consisting of one or more numeric digits with an optional decimal point. These digits are followed by the letter E, followed by an optionally signed exponent representing an integral power of 10. For example:

3.12E-11

See also constant.

floating-point scale

The format of an arithmetic datum in which the datum is a rational number with a fractional part and an exponent part. See also scale.

format item

An element of a format list, in edit-directed input/output. It can specify either the representation of a data item or its positioning in the stream. See also edit-directed input/output.

format list

A list of format items that controls the transmission of data to or from a stream I/O file during the execution of an edit-directed GET or PUT statement. See also edit-directed input/output.

fully qualified reference

A qualified reference that includes the name of each containing structure from the major structure down to the referenced member. See also partially qualified reference, qualified reference, structure.

function

A procedure that returns a value. Also called function procedure. See also procedure, subroutine.

identifier

See name.

infix operator

Any operator placed between expressions. See also operator.

integer constant

A constant consisting only of one or more numeric digits. For example:

25

See also constant.

internal procedure block

See internal procedure.

internal procedure

A procedure whose entry name is within the scope of an encompassing block. Also called internal procedure block, nested procedure, subprocedure. See also procedure, external procedure.

key

A data item that identifies a record that is directly accessed. See also record file, direct access.

keyword

An identifier that has a language-defined or implementation-defined meaning to the compiler when used in particular contexts.

label

A name that identifies a statement other than a PROCEDURE or ENTRY statement. See also statement.

level number

A decimal constant in the declaration of a structure that specifies the hierarchical position of a variable in the structure. See also structure.

list-directed input/output

Transmission of data to or from a stream file without a format-list. See also data-directed input/output, edit-directed input/output, stream file.

mode

A characteristic of arithmetic data; in PL/I, either REAL or COMPLEX. See also arithmetic data.

name

A string of up to 32 characters, which can be the alphanumeric characters, \$, #, and the underscore (_) character. The first character of a name must be a letter. Also called identifier. See also scope of a name.

nested procedure

See internal procedure.

null character string

A character string of zero length. See also character string.

null pointer

A pointer value produced by the NULL built-in function. A null pointer is a unique value that addresses no variable and is used to indicate that a pointer variable does not currently address anything. See also pointer.

null statement

An empty statement, consisting only of a semicolon (;). It has no effect. See also statement.

null string

A zero-length character-string or bit-string. See also string data.

on-condition

A condition specified in an ON statement. See also condition.

on-unit

A BEGIN block or statement (other than PROCEDURE, DO, END, DECLARE or FORMAT) that describes an action to be taken upon the occurrence of an on-condition. See also condition, on-condition.

operand

A part of an expression that is not an operator. It may be a constant, a variable reference, a function reference, a built-in function reference, or another expression.

operator

A symbol specifying an operation. The operators used in PL/I are as follows: +, -, *, /, **, =, ^=, <, >, <=, >=, ^<, ^>, &, ^, | (or !), and || (or !!). See also arithmetic operator, bit-string operator, comparison operator, infix operator, prefix operator, string operator.

parameter

An identifier in a PROCEDURE statement for which a value is substituted by an invoking PROCEDURE reference. See also argument.

partially qualified reference

A qualified reference that is unique, but from which one or more of the names of containing structures have been omitted. See also fully qualified reference, qualified reference, structure.

pictured data

Values that can be either numeric or character, that are represented by means of precise specifications for the positions of characters, and upon which computations can be performed. See also arithmetic data, string data.

POINTER

An attribute specifying the declared identifier as a pointer variable that contains the address of a based storage datum. See also based storage.

pointer qualification

Identification of a based variable by the use of a pointer followed by the pointer qualification symbol, such as Q->ALPHA. Pointer-valued functions and pointer-valued built-in functions may also be used as pointer qualifiers.

pointer qualification symbol

The character sequence ->, signifying that the pointer value on the left of -> gives the address to be used with the BASED variable reference on the right.

pointer value

A value whose data type is POINTER.

pointer variable

A variable, declared with the POINTER data type attribute, whose value is an address in memory.

precision

The number of significant binary or decimal digits maintained for the value of an arithmetic variable and, optionally, the number of those digits that are fractional. As an attribute, it is specified by a parenthesized decimal number or by a parenthesized pair of numbers separated by a comma. See also arithmetic data.

prefix operator

Any of the operators +, -, and ^, placed to the left of an expression. See also operator.

procedure

A sequence of statements beginning with a PROCEDURE statement and terminated by the matching END statement. Also called procedure block. See also block, function, subroutine, internal procedure, external procedure.

PROCEDURE block

See procedure.

PROCEDURE block name

See procedure name.

PROCEDURE block reference

See procedure reference.

procedure name

A name designating the entry point to a procedure (the name of the PROCEDURE statement). Also called PROCEDURE block name.

procedure reference

Invocation of a procedure by a CALL statement or function reference; any reference that is followed by an argument list consisting of a parenthesized list of expressions separated by commas, or followed by an empty argument list (). (An empty argument list may be omitted in the procedure reference of a CALL statement.) See also argument.

pseudovisible

A built-in function used on the left side of an assignment statement. In each case, the built-in function acts as if it were a variable. The pseudovisibles are IMAG, ONCHAR, ONSOURCE, PAGENO, REAL, STRING, SUBSTR, and UNSPEC. See also built-in function.

qualified name

See qualified reference.

qualified reference

A reference to a variable in a structure, consisting of a sequence of names written left to right in order of increasing level numbers and separated by periods. Blanks may be inserted around the periods. The sequence must include sufficient names to make the reference unique. See also level number, fully qualified reference, partially qualified reference, structure.

record file

A file organized into a set of discrete records that are either accessible sequentially or accessible directly by key. See also file, direct access, sequential access, stream file.

recursive PROCEDURE block

See recursive procedure.

recursive procedure

A procedure that can call itself. The idea of recursion is derived from the mathematical concept of a recursive function, which is a function that may be defined in terms of itself. For example, $N!$ (N factorial) is defined as

$$\begin{aligned} N! &= 1 \text{ for } N = 1 \\ N! &= N * (N - 1)! \text{ for } N > 1 \end{aligned}$$

Also called recursive PROCEDURE block.

reference

The use, in a context other than in a declaration, of a name, together with any subscripts, pointer qualifier, or structure names necessary to indicate the object of the reference. References to procedures or built-in functions may also contain an argument list. In order to determine the meaning of the reference, the compiler searches for the declaration of the name. This search resolves the reference by associating it with a declaration of the name. See also qualified reference, simple reference, subscripted reference.

scalar

A single data item. See also aggregate.

scale

The system of mathematical notation used to represent an arithmetic value; in PL/I, either fixed-point (FIXED) or floating-point (FLOAT). See also arithmetic data, fixed-point scale, floating-point scale.

scope of a condition prefix

The region of a program to which a condition prefix applies.

The scope of a condition prefix applied to a BEGIN or PROCEDURE statement is the entire block.

The scope of a condition prefix applied to a DO statement is the DO statement itself. The condition prefix does not apply to the entire DO group.

The scope of a condition prefix applied to an IF statement is only the expression immediately following the IF keyword. The condition prefix does not apply to the THEN or ELSE clauses of the IF statement.

The scope of a condition prefix applied to an ON statement does not include the on-unit.

In all other cases, the condition prefix applies to the entire statement.

See also condition prefix.

scope of a name

The region of a program over which a name (identifier) is known and can be referred to. The scope of a name includes the block in which it is declared and all blocks contained within that block, except those blocks in which the name is redeclared. See also name.

scope of a declaration

The region of a program to which a particular declaration of a name (identifier) applies. The scope of a declaration includes the block in which it appears and all contained blocks, except blocks in which the name has been redeclared.

separator

A character recognized by the compiler as a delimiter of program elements. Separators are the following characters: () , . ; : and the blank character.

sequential access

A method of storing records in files such that the file must be searched from the beginning to find a particular record. See also direct access, record file.

simple reference

A reference to a name without any subscripts, pointer reference, or argument list. See also reference, qualified reference, subscripted reference.

stack frame

A block of storage allocated on a stack used to hold information that is unique to each procedure activation, such as the location to which control should return from the procedure activation.

statement

A sequence of tokens (elements) ending with a semicolon. All statements, except the assignment statement, begin with a keyword that identifies the purpose of the statement. See also compound statement, token.

statement identifier

A keyword naming a statement. For example, DO is the statement identifier of the DO statement. See also keyword.

statement label

A name identifying a statement. See also label.

static storage

Storage that is allocated before execution of the program and is released at program termination. See also dynamic storage, storage class.

storage class

An attribute of a variable that determines how and when storage is allocated for the variable. See also allocation, automatic storage, based storage, controlled storage, dynamic storage, static storage.

stream file

A file containing a sequence of characters organized into lines. See also file, record file.

string data

Values consisting of a sequence of bits or characters on that string operations are allowed. See also bit string, character string, arithmetic data, pictured data.

string operator

The operator || (or !!) denoting concatenation. See also operator.

structure

A hierarchically ordered set of variables that may be of different data types. See also level number, qualified reference.

subprocedure

See internal procedure.

subroutine

A procedure that does not return a value. Also called subroutine procedure. See also procedure, function.

subscript

An integer, or an expression evaluating to an integer, used to reference an array element. Elements of an array are referenced using as many subscripts as the array has dimensions. See also array, simple reference.

subscripted reference

A reference to a name that has been declared as an array, followed by a parenthesized list of subscript expressions. See also reference, qualified reference, simple reference.

substructure

A structure that is itself a member of another structure. See also structure.

termination of BEGIN block

The passing of control out of a BEGIN block, accomplished by execution of the END statement for the block or of a nonlocal GO TO statement. See also BEGIN block.

termination of PROCEDURE block

The passing of control out of a PROCEDURE block, accomplished by execution of a RETURN statement, of the END statement for the block, or of a nonlocal GO TO statement. See also procedure.

token

The basic element of the PL/I language. A token can be a name, a constant, a punctuation symbol, a comment, or a compile-time text-modification statement. See also statement.

variable

A named object that is capable of holding values. Each variable has two properties: data type and storage class.

H

Use of FORMS with PL/I

The following briefly summarizes the use of FORMS with PL/I syntax. For more information, consult the FORMS Programmer's Guide, PDR3040-163P. You should make sure to declare FORM\$I as an entry, to call this procedure before doing any I/O, and to open a file with the TITLE option, in any of the three ways described below.

- TITLE('@TTY -FORMS')

Allows both input and output to be performed via the file designator. The -FORMS option tells the compiler that FORMS will be used. This MUST be in the title declaration as shown.

- TITLE('SYSPRINT -FORMS')

Allows FORMS to be done in an OUTPUT mode only.

- TITLE('SYSIN -FORMS')

Allows FORMS to be done in an INPUT mode only.

The recommended manner is the first.

From then on, use normal ASCII I/O statements. You may use either PUT EDIT or PUT LIST. Since FORMS is usually used in a formatted fashion, PUT EDIT is recommended; it gives the user control over how the formatting is done. With PUT LIST, the compiler determines the formatting under predefined rules.

PL/I Reference Guide

In addition, at load time, you must remember to load VFORMS before invoking the PL1 or PL1G library.

Below is an example of the use of FORMS in a PL/I procedure:

```
X: PROC;
  DCL FORM$ I ENTRY;
  DCL F FILE STREAM;
  DCL G FILE OUTPUT STREAM;
  DCL A FIXED BIN;
  DCL H FILE OUTPUT STREAM;

  CALL FORM$ I;
  OPEN FILE(H) TITLE('@TTY -FORMS');
  PUT FILE(H) EDIT('##INVOKE DS1') (A(12));
  PUT FILE(H) EDIT('##CLEAR') (A(7));
  END;
```

I

Using SEG

This appendix tells how to use Prime's older load utility, SEG.

LOADING AND EXECUTING PL/I PROGRAMS

To load and execute an object program produced by the PL/I compiler, use the following steps if the object filename ends in .BIN.

1. Invoke Prime's loader with the SEG -LOAD command. SEG responds with the prompt \$.
2. Use the LOAD subcommand with either program.BIN or simply program.
3. Use the LOAD subcommand with the names of any subroutines that were compiled separately, loading them in the order called.
4. Use the LI command with any necessary Prime libraries.
5. Use LI with the PL/I library (PLLIB).
6. Use LI alone to load other system library routines. If all references are resolved, the message LOAD COMPLETE appears. If this message is not displayed, enter MAP 3 for a display of routines that are missing.

7. Either leave SEG and store the newly created runfile with QUIT, or start execution with EXEC.

Subsequent executions of the runfile can be started with

SEG program

For example, if your source program is stored in a PRIMOS file named MYPROG.PL1, you can compile it with the command

```
PL1 MYPROG
```

This compilation produces an object file named MYPROG.BIN. Assume that no subroutines or special libraries (such as the sort library) are needed. Load and execute the program with a dialog such as the following:

```
OK, SEG -LOAD  
[SEG rev 19.4]  
$ LO MYPROG  
$ LI PL1LIB  
$ LI  
LOAD COMPLETE  
$ EXEC
```

For subsequent executions, enter:

```
SEG MYPROG
```

INDEX

Index

Symbols

! (See |)

!! (See ||)

-> 7-9

\$ (dollar sign),
in pictured-numeric
specification, 5-50

& (ampersand),
logical operator, 6-35
logical operator, table, 6-35

' (apostrophe),
specification of BIT constants,
5-24
specification of CHARACTER
constants, 5-19
within CHARACTER constants,
5-22

* (asterisk),
in INITIAL values, 7-27
in parameter extent
expressions, 8-25
in pictured-numeric
specification, 5-45
in RETURNS descriptor, 8-20

* (asterisk) (continued)
infix operator, 6-27
infix operator, table, 6-27
initializing arrays, 5-74
specifying array cross
sections, 5-62

** (double asterisk),
infix operator, 6-30

+ (plus sign),
in pictured-numeric
specification, 5-35, 5-46
infix operator, 6-25
infix operator, table, 6-26
prefix operator, 6-36
prefix operator, table, 6-37

, (comma),
in pictured-numeric
specification, 5-38
in structure declarations,
5-63

- (minus sign),
in pictured-numeric
specification, 5-35, 5-46
infix operator, 6-25
infix operator, table, 6-26
prefix operator, 6-36
prefix operator, table, 6-37

PL/I Reference Guide

- . (period),
 - in pictured-numeric specification, 5-38
- / (slash),
 - in pictured-numeric specification, 5-38
 - infix operator, 6-28
 - infix operator, table, 6-29
- < (less than),
 - comparison operator, 6-32
- <= (less than or equal to),
 - comparison operator, 6-32
- = (equal sign),
 - comparison operator, 6-32
 - in assignment statement, 4-5
- > (greater than),
 - comparison operator, 6-32
- >= (greater than or equal to),
 - comparison operator, 6-32
- ^ (caret),
 - in Prime EDITOR, 6-4
 - prefix operator, 6-37
 - prefix operator, table, 6-37
- ^< (not less than),
 - comparison operator, 6-32
- ^= (not equal),
 - comparison operator, 6-32
- ^> (not greater than),
 - comparison operator, 6-32
- | (vertical bar),
 - logical operator, 6-35
 - logical operator, table, 6-36
- || (double vertical bar),
 - infix operator, 6-31

Numbers

-64V compiler option, 2-4

- 9,
 - in pictured-numeric specification, 5-32
 - in pictured-string specification, 5-30

A

- A,
 - in pictured-string specification, 5-30
- A data format item,
 - input, introduction, 11-21
 - input, specifications, 11-66
 - input, table, 11-67
 - output, introduction, 11-8
 - output, specifications, 11-47
 - output, table, 11-47

ABS built-in function, 14-4,
14-14
introduction, 4-30

ACOS built-in function, 14-15

ADD built-in function, 14-16

Addition operator, 6-25

ADDR built-in function, 7-22,
14-3, 14-16

AFTER built-in function, 14-17

Aggregates,

- and GET LIST, 11-15
- and PUT DATA, 11-5
- and PUT EDIT, 11-11
- and PUT LIST, 11-4
- arguments to built-in functions, 14-3
- data conversion, 6-43
- in expressions, 6-43
- introduction, 4-42
- promotion, introduction, 6-1
- returning, from function procedures, 8-16

ALIGNED attribute, 5-66

- ALLOCATE statement, 7-7
 errors in, 13-26
 IN option, 7-15
 raising AREA condition, 13-26
 SET option, 7-8
 syntax, 7-19
- ALLOCATION built-in function,
 7-7, 14-3, 14-18
- ALLOCN (See ALLOCATION)
- ALLOW_PRECONNECTION compiler
 option, 2-4
- And,
 logical operation, 6-35
- ANSI option, 11-29
- ANSI standard, 1-1
- Apostrophes,
 specification of BIT constants,
 5-24
 specification of CHARACTER
 constants, 5-19
 within CHARACTER constants,
 5-22
- APPEND option,
 with RECORD files, 12-9, 12-24
 with STREAM files, 11-29
- APRE (See -ALLOW_PRECONNECTION)
- AREA condition, 13-26
- AREA variables, 7-14
 (See also Noncomputational data
 types)
 contextual declaration, 9-20
- Arguments,
 arrays, 8-29
 dummy, 8-25
 for function procedures, 8-13
 in CALL statement, 8-10
 parameters, relation to, 8-21
- Arguments to built-in functions,
 14-2
 aggregate, 14-3
 converted precision, 14-4
- Arguments to built-in functions
 (continued)
 derived data type, 14-4
 specifying precision, 14-5
- Arithmetic built-in functions,
 ABS, 14-14
 ADD, 14-16
 BINARY, 14-24
 CEIL, 14-30, 14-91
 classification and summary,
 14-7
 DECIMAL, 14-39
 DIVIDE, 14-41
 FIXED, 14-46
 FLOAT, 14-47
 FLOOR, 14-48, 14-91
 introduction, 4-30
 MAX, 14-57
 MIN, 14-58
 MOD, 14-59
 MULTIPLY, 14-62
 PRECISION, 14-68
 ROUND, 14-72
 SIGN, 14-74
 SUBTRACT, 14-81
 TRUNC, 14-90
- Arithmetic data, (See also
 Computational data types;
 Data conversion)
 COMPLEX, 5-14
 constants, summary, 5-17
 conversion, 6-38
 conversion, introduction, 4-27
 declaring variables,
 introduction, 4-23
 defined, 5-3
 FIXED BINARY, 5-10
 FIXED DECIMAL, 5-5
 FLOAT BINARY, 5-13
 FLOAT DECIMAL, 5-9
 introduction, 4-22, 5-4
 variables, declaring, 5-16
- Arithmetic operators,
 introduction, 4-11, 6-3
- Array subscripts, 5-57
 data conversion, 6-10
 error checking with -RANGE,
 2-10
 errors, 13-13, 13-37

Array subscripts (continued)
introduction, 4-43
SUBSCRIPTRANGE condition,
13-13, 13-37

Array-handling built-in
functions,
classification and summary,
14-11
DIMENSION, 8-30, 14-3, 14-40
DOT, 14-3, 14-42
HBOUND, 8-29, 14-3, 14-49
LBOUND, 8-29, 14-3, 14-52
PROD, 14-3, 14-69
SUM, 14-3, 14-82

Arrays, (See also Aggregate
promotion)
aggregate promotion, 6-44
as arguments and parameters,
8-29
bounds, 5-59
cross sections, 5-62
declaring with DEFINED
attribute, 5-68
in expressions, 6-43
initializing, 5-73, 7-26
internal representation, C-10
introduction, 4-43
iSUB defining, 5-70
multi-dimensional, 5-59
of file variables, 12-34
of structures, 5-65
of structures, aggregate
promotion, 6-46
of structures, initializing,
5-74
of structures, introduction,
4-45
one-dimensional, 5-56
returning, from functions,
8-16

ASCII character set, B-1

ASCII-8 character set, B-2

ASIN built-in function, 14-19

Assignment statement,
introduction, 4-5

Asterisk,
in INITIAL values, 7-27
in parameter extent
expressions, 8-25
in pictured-numeric
specifications, 5-45
in RETURNS descriptor, 8-20
initializing arrays, 5-74
specifying array cross
sections, 5-62

ATAN built-in function, 14-20
illustration, 14-21

ATAND built-in function, 14-22

ATANH built-in function, 14-22

Attributes, (See also File
attributes)

ALIGNED, 5-66
BUILTIN, 14-2
defaults, overriding, 5-75
DEFINED, 5-67, 7-24
EXTERNAL, 7-30, 8-34
GENERIC, 8-39
INITIAL, 5-72
INTERNAL, 7-30
LIKE, 5-71, 9-13
POSITION, 5-69
UNALIGNED, 5-66

AUTOMATIC storage class, 7-4
defined, 7-3
in recursive procedures, 7-6
variables in extent and INITIAL
expressions, 7-28

B

B,
in pictured-numeric
specification, 5-38
specification of BINARY
constants, 5-10, 5-13
specification of BIT constants,
5-24

-B (See -BINARY)

- B data format item,
 - input, specifications, 11-67
 - input, table, 11-68, 11-69
 - output, specifications, 11-48
 - output, table, 11-49, 11-50
- Base of arithmetic data,
 - defined, 5-4
 - derived common, 6-18
 - derived common, table, 6-19
 - introduction, 4-23
- BASED storage class, (See also POINTER data)
 - AREA and OFFSET variables, 7-14
 - defined, 7-3
 - linked lists and, 7-10
 - locate mode input/output and, 12-18
 - POINTER variables and, 7-8
 - REFER option and, 7-29
 - STATIC storage and, 7-10
 - variables in extent and INITIAL expressions, 7-29
- Batch job program environment, 1-6
- BEFORE built-in function, 14-23
- BEGIN statement, (See also Blocks)
 - block invocation and termination, 10-28
 - scope of condition prefix, 13-16
- BIG compiler option, 2-4
- BIN (See BINARY)
- BINARY built-in function, 14-24
- BINARY compiler option, 2-4
- BINARY data, (See also Base of arithmetic data)
 - FIXED, 5-10
 - FLOAT, 5-13
- BIND command, 3-1
- BIND commands,
 - FILE, 3-2
 - HELP, 3-3
 - LIBRARY, 3-2
 - LOAD, 3-2
 - MAP, 3-3
 - QUIT, 3-3
- BIT built-in function, 14-26
- BIT data, (See also CHARACTER data; String data)
 - ALIGNED attribute and, 5-66
 - as logical variables, 5-25
 - assignment errors, 13-36
 - comparison operators and, 6-4
 - constants, 5-26
 - constants, table, 5-26
 - conversion from/to CHARACTER, 6-41
 - conversion from/to numeric, 6-40
 - data conversion, 6-24
 - discussion, 5-23
 - internal representation, C-7
 - NONVARYING, 5-24
 - null string, 5-26
 - octal notation, 5-27
 - other number bases, 5-27
 - other number bases, table, 5-28
 - string overlay defining, 5-70
 - VARYING, 5-25
- Blocks, (See also Declarations; DO groups; Scope rules)
 - active and inactive, 10-30
 - classification, 9-3, 10-27
 - containment of declarations, 9-23
 - dynamic storage area (DSA), 10-31, 10-39
 - environmental block invocation, 10-32
 - epilogue execution, 10-28
 - inheriting variables, 10-31
 - introduction, 4-50, 9-1
 - invocation, 10-27
 - invocation and termination, summary, 10-47
 - invoking block invocation, 10-32
 - multiple closure END statements, 9-5

Blocks (continued)
 nesting, 9-4
 on-units, implementation,
 10-47
 prologue execution, 10-28
 SELECT, 10-25
 storage management, 10-28
 structure, 10-32
 structure, static and dynamic,
 10-29
 termination, 10-22, 10-28

BOOL built-in function, 14-27
 table, 14-27, 14-28

Boolean expressions (See BIT
 data; Logical expressions)

Built-in functions, (See also
 Arithmetic built-in
 functions; Array-handling
 built-in functions;
 Condition-handling built-in
 functions; Mathematical
 built-in functions;
 Miscellaneous built-in
 functions; Storage-handling
 built-in functions;
 String-handling built-in
 functions)

ABS, 14-4, 14-14
 ACOS, 14-15
 ADD, 14-16
 ADDR, 7-22, 14-16
 AFTER, 14-17
 aggregate arguments, 14-3
 ALLOCATION, 7-7, 14-18
 arguments, 14-2
 arguments that specify
 precision, 14-5
 arithmetic, 14-7
 array-handling, 14-3, 14-11
 ASIN, 14-19
 ATAN, 14-20
 ATAND, 14-22
 ATANH, 14-22
 BEFORE, 14-23
 BINARY, 14-24
 BIT, 14-26
 BOOL, 14-27
 BYTE, 14-29
 CEIL, 14-30, 14-91
 CHARACTER, 14-31

Built-in functions (continued)
 classification and summary,
 14-6
 COLLATE, 14-32
 COMPLEX, 14-32
 condition-handling, 14-12
 CONJG, 14-33
 contextual declaration, 9-17
 converted precision, 14-4
 COPY, 14-34
 COS, 14-35
 COSD, 14-35
 COSH, 14-36
 data conversion, 6-16, 6-42,
 14-4
 DATE, 14-2, 14-36
 DECAT, 14-37
 DECIMAL, 14-39
 defined, 14-1
 DIMENSION, 8-30, 14-40
 DIVIDE, 14-41
 DOT, 14-42
 EMPTY, 14-43
 ERF, 14-43
 ERFC, 14-44
 EVERY, 14-45
 EXP, 14-46
 expressions and, 6-8
 FIXED, 14-46
 FLOAT, 14-47
 FLOOR, 14-48, 14-91
 general rule for aggregates,
 14-3
 HBOUND, 8-29, 14-49
 HIGH, 14-50
 IMAG, 14-51
 INDEX, 4-38, 14-51
 input/output-related, list of,
 12-33
 introduction, 4-28
 LBOUND, 8-29, 14-52
 LENGTH, 14-53
 LINENO, 11-33, 14-54
 LOG, 14-54
 LOG10, 14-55
 LOG2, 14-56
 LOW, 14-56
 mathematical, 14-8
 MAX, 14-4, 14-57
 MIN, 14-58
 miscellaneous, 14-13
 MOD, 14-59
 MULTIPLY, 14-62
 NULL, 7-13, 14-63

Built-in functions (continued)

OFFSET, 7-18, 14-63
 on-units and, 13-40
 ONCHAR, 13-6, 14-64
 ONCODE, 13-40, 14-64, F-1
 ONFIELD, 14-65
 ONFILE, 14-65
 ONKEY, 14-66
 ONLOC, 14-66
 ONSOURCE, 13-6, 14-67
 PAGENO, 11-33, 14-67
 POINTER, 7-16, 14-68
 PRECISION, 14-68
 Prime extensions, 14-29,
 14-70, 14-77
 PROD, 14-69
 RANK, 14-70
 REAL, 14-71
 REVERSE, 14-71
 ROUND, 14-72
 SIGN, 14-74
 SIN, 14-75
 SIND, 14-76
 SINH, 14-76
 SIZE, 14-77
 SOME, 14-78
 SQRT, 14-78
 storage-handling, 14-3, 14-12
 STRING, 14-79
 string-handling, 14-9
 SUBSTR, 4-37, 13-35, 14-80
 SUBTRACT, 14-81
 SUM, 14-82
 TAN, 14-83
 TAND, 14-84
 TANH, 14-85
 terminology, 4-29
 TIME, 14-85
 TRANSLATE, 14-86
 TRIM, 14-89
 TRUNC, 14-90
 UNSPEC, 14-92
 VALID, 14-93
 VERIFY, 14-94
 without arguments, 14-2

 BUILTIN attribute, 14-2
 declaring, 14-2

 BY NAME option, 5-65

 BY option, 10-6

 BYTE built-in function, 14-29

C

C data format item,
 input, specifications, 11-70
 input, table, 11-70
 output, introduction, 11-8
 output, specifications, 11-51
 output, table, 11-52

 CALL statement,
 arguments, 8-10
 introduction, 8-3

 Case selection, 10-25

 CEIL built-in function, 14-30,
 14-91
 introduction, 4-30

 CHAR (See CHARACTER)

 CHARACTER built-in function,
 14-31

 CHARACTER data, (See also BIT
 data; String data;
 String-handling built-in
 functions)
 ALIGNED attribute and, 5-67
 apostrophes within constants,
 5-22
 assignment errors, 13-36
 comparison, 4-36
 computation with, 6-20
 concatenation, 4-36
 constants, introduction, 4-7,
 4-34
 constants, rules for forming,
 5-21
 constants, table, 5-22
 conversion from/to BIT, 6-41
 conversion from/to numeric,
 6-39
 data conversion, 6-24
 declaring, 4-33
 initializing, 5-74
 internal representation, C-6
 introduction, 4-32
 NONVARYING, discussion, 5-19
 null string, 4-34, 5-22
 operations, 4-35
 output file representation,
 11-33
 repetition factors, 5-23

CHARACTER data (continued)

replication factors, in INITIAL values, 7-27
string overlay defining, 5-68
substrings, 4-37
VARYING, 5-20
VARYING, internal representation, C-7

CLOSE statement,

effect on file attributes, 12-31
omission of, effect, 12-31
with sequential access files, 12-7

Code size,

maximum, 1-4

COLLATE built-in function, 14-32

Colon,

as range specifier in GENERIC declarations, 8-40
in array-bound spec, 5-59

COLUMN control format item,

input, introduction, 11-22
input, specifications, 11-71
output, introduction, 11-10
output, specifications, 11-52

Comma,

in pictured-numeric specifications, 5-38
in structure declarations, 5-63

Comments,

syntax, 4-10

Comparison operators,

and CHARACTER data, 4-36
discussion, 6-32
introduction, 4-15, 6-3
table, 6-3, 6-32

Compiler options,

-64V, 2-4
-ALLOW_PRECONNECTION, 2-4
-BIG, 2-4
-BINARY, 2-4
-COPY, 2-5
-DEBUG, 2-5

Compiler options (continued)

-ERRLIST, 2-5
-ERRTTY, 2-6
-EXPLIST, 2-6
-FRN, 2-6
-FULL_HELP, 2-6
-FULL_OPTIMIZE, 2-7
-HELP, 2-7
-INPUT, 2-7
introduction, 2-2
-LCASE, 2-7, 4-8
-LISTING, 2-8
-MAP, 2-8
-MAPWIDE, 2-8
-NESTING, 2-9
-OFFSET, 2-9
-OPTIMIZE, 2-9
-OVERFLOW, 2-10
-PRODUCTION, 2-10
-RANGE, 2-11
-SILENT, 2-11
-SOURCE, 2-11
-SPACE, 2-12
-STATISTICS, 2-12
-STORE_OWNER_FIELD, 2-12
-STRINGSIZE, 2-13
table, 2-14
-TIME, 2-13
-UPCASE, 2-13
-XREF, 2-13

Compiler-directing statements,

%INCLUDE, 10-53
%LIST, 10-53
%NOLIST, 10-53
%PAGE, 10-53
%REPLACE, 10-53

Compiling programs,

concepts, 8-31
with Prime PL/I compiler, 2-1

COMPLEX built-in function, 14-32

COMPLEX data, (See also Mode of arithmetic data)

constants, table, 5-15
discussion, 5-14
FIXED BINARY, internal representation, C-4
FIXED DECIMAL, internal representation, C-4
FLOAT BINARY, internal representation, C-5

- COMPLEX data (continued)
 FLOAT DECIMAL, internal representation, C-5
- Computational data types,
 introduction, 5-3
 scalar conversion rules, 6-38
- Concatenation, 6-5
 introduction, 4-36
 operator, 6-31
- COND (See CONDITION)
- CONDITION condition, 13-20,
 13-26
 and SIGNAL statement, 13-20
- Condition prefixes, 13-13
 introduction, 4-50
 overriding, 13-17
 scope rules, 13-16
 with BEGIN statement, 13-16
 with DO statement, 13-16
 with IF statement, 13-17
 with ON statement, 13-17
 with PROCEDURE statement,
 13-16
- Condition-handling built-in
 functions,
 and SIGNAL statement, 13-41
 classification and summary,
 14-12
 ONCHAR, 13-6, 14-64
 ONCODE, 13-40, 14-64, F-1
 ONFIELD, 14-65
 ONFILE, 14-65
 ONKEY, 14-66
 ONLOC, 14-66
 ONSOURCE, 13-6, 14-67
 table, 13-42
- Conditions, (See also ON
 statement; On-units)
 AREA, 13-26
 CONDITION, 13-20, 13-26
 condition prefixes, 13-13
 contextual declaration, 9-19
 CONVERSION, 13-6, 13-15, 13-27
 debugging with, 13-17
 default states, 13-16
 defined, 13-1
 enabling, 13-13
- Conditions (continued)
 enabling and disabling, 13-16
 ENDFILE, 11-38, 13-4, 13-27
 ENDPAGE, 11-33, 11-38, 13-28
 ERROR, 13-9, 13-29
 FINISH, 13-29
 FIXEDOVERFLOW, 13-30, 13-32
 flow chart, 13-25
 input/output, list of, 12-32
 introduction, 4-50
 KEY, 13-31
 list of, 13-24
 NAME, 13-31
 NOCONVERSION, 13-15
 NOFIXEDOVERFLOW, 13-30
 NOOVERFLOW, 13-32
 NOSTRINGSIZE, 13-36
 NOUNDERFLOW, 13-39
 NOZERODIVIDE, 13-40
 OVERFLOW, 13-32
 PRIMOS condition-handling
 mechanism, 1-8
 raising artificially, 13-19
 RECORD, 13-33
 SIGNAL statement, 13-19
 SIZE, 13-33
 SNAP option, 13-20
 stack frame representation,
 D-2
 standard system action, 13-2
 STORAGE, 13-34
 STRINGRANGE, 13-35
 STRINGSIZE, 13-36
 SUBSCRIPTRANGE, 13-13, 13-37
 TRANSMIT, 13-37
 UNDEFINEDFILE, 13-38
 UNDERFLOW, 13-39
 user-defined, 13-20
 ZERODIVIDE, 13-40
- CONJG built-in function, 14-33
- CONSTANT file attribute, 12-33
- Constants,
 arithmetic, summary, 5-17
 CHARACTER, 4-7, 4-34
 data types, introduction, 5-2
 declaring in statement labels,
 9-14
 ENTRY, 7-31, 9-14
 FILE, 7-31, 9-18
 FORMAT, 7-31, 9-14

- Constants (continued)
 - introduction, 4-9
 - LABEL, 7-31, 9-14
- Control flow (See Flow of control)
- CONTROLLED storage class, 7-6
 - defined, 7-3
 - EXTERNAL attribute and, 7-30
 - variables in extent and INITIAL expressions, 7-28
- CONV (See CONVERSION condition)
- Conversion (See Data conversion)
- CONVERSION condition, 13-27
 - disabling, 13-15
 - use of, 13-6, 13-15
- COPY built-in function, 14-3, 14-34
- COPY compiler option, 2-5
- COPY input option,
 - introduction, 11-14
 - specifications, 11-72
- COS built-in function, 14-35
- COSD built-in function, 14-35
- COSH built-in function, 14-36
- CPLX (See COMPLEX)
- CR,
 - in pictured-numeric specification, 5-35
- Cross-reference listing (See -XREF)
- CTLASA option, 12-25

- D
- DAM files,
 - changing, 12-12
 - creating sequentially, 12-10
- DAM files (continued)
 - expanding, 12-13
 - introduction, 12-3
 - KEY option, 12-12
 - MIDASPLUS files, compared to, 12-14
 - reading with KEYED DIRECT INPUT, 12-10
 - updating with KEYED DIRECT UPDATE, 12-12
- DAM option, 12-24
- Data conversion,
 - among arithmetic variables, introduction, 4-27
 - arrays, 6-44
 - arrays of structures, 6-46
 - BIT to CHARACTER, 6-41
 - BIT to numeric, 6-41
 - built-in functions, 6-42, 14-4
 - CHARACTER to BIT, 6-41
 - CHARACTER to numeric, 6-40
 - comparison expressions and, 6-32
 - computational data types, rules, 6-38
 - converted precision, 6-21
 - converted precision, table, 6-23
 - derived common base, scale, and mode, 6-13
 - derived common string type, 6-24
 - discussion, 6-9
 - from CHAR, use of CONVERSION condition, 13-6, 13-15, 13-27
 - implicit conversion, 6-12, 6-20
 - intermediate targets, 6-10
 - introduction, 6-1
 - numeric to BIT, 6-40
 - numeric to CHARACTER, 6-39
 - numeric to numeric, 6-38
 - structures, 6-45
 - with PICTURE data, 6-41
- DATA option, (See also GET DATA statement; PUT DATA statement)
 - input, specifications, 11-72
 - output, specifications, 11-53

- Data sets (See Files)
- Data types, (See also Arithmetic data; PICTURE data; String data)
 arrays and structures, 5-56
 classification, 5-2
 default, introduction, 4-25
 introduction, 5-1
- DATE built-in function, 14-2, 14-36
- DB,
 in pictured-numeric specification, 5-35
- DEBUG compiler option, 2-5
- Debugging programs, (See also -DEBUG)
 -PRODUCTION compiler option, 2-10
 PUT DATA statement, 11-5
 Source Level Debugger, 1-7
 -STORE_OWNER_FIELD compiler option, 2-12
 use of conditions, 13-17
 use of SIGNAL statement, 13-20
- DEC (See DECIMAL)
- DECAT built-in function, 14-37
 table, 14-38
- DECIMAL built-in function, 14-39
- DECIMAL data, (See also Base of arithmetic data)
 FIXED, 5-5
 FLOAT, 5-9
- Declarations, (See also Blocks)
 containment, 9-23
 contextual, 9-16, 9-26
 explicit, scope of, 9-23, 9-25
 explicit, types of, 9-14
 factored, 9-9
 immediate containment, 9-24
 implicit, 9-21, 9-26
 multiple, 9-8, 9-27
 of arithmetic variables, 5-16
 of external procedures, IBM
 versus ANS practice, 9-20
- Declarations (continued)
 resolving references, 9-27
 scope rules, 9-21
 structures, 9-7
- DECLARE statement,
 file attributes and, 12-20
 for arithmetic variables, 4-23
 for external procedures, 8-34
 GENERIC attribute, 8-39
 INITIAL attribute, 7-2, 7-26
 introduction, 4-23, 9-6
 LIKE attribute, 9-13
 specifying data types with, 5-2
 with STREAM files, 11-25
- DEFAULT statement, 5-75
 compiler application, 5-78
 ERROR option, 5-79
 format, 5-77
 NONE option, 5-79
 P-constants and, 5-79
 SYSTEM option, 5-79
- Defaults,
 alignment, 5-67
 arithmetic data attributes, 5-16
 BIT data, 5-25
 CHARACTER data, 5-20
 data types, introduction, 4-25
 in implicit declarations, 9-21
 overriding with DEFAULT statement, 5-75
 scope, for FILE and ENTRY constants, 7-31
 scope, for variables, 7-30
 storage type, 7-3
- DEFINED attribute, 7-24
 BIT string overlay defining, 5-70
 CHARACTER string overlay defining, 5-68
 defined, 7-3
 iSUB defining, 5-70, 7-25
 POSITION attribute and, 5-69
 simple defining, 5-67
- DELETE statement, 12-14, 12-16

- DEVICE option,
 - with RECORD files, 12-24
 - with STREAM files, 11-29
 - Devices, input/output,
 - table, 12-25
 - DFT (See DEFAULT)
 - DIM (See DIMENSION)
 - DIMENSION built-in function,
 - 8-30, 14-3, 14-40
 - Dimensioned variables (See Arrays)
 - Direct access files, (See also DAM files; MIDASPLUS files; Sequential access files)
 - introduction, 12-2
 - keys, 12-4
 - DIRECT file attribute, 12-22
 - directories,
 - specifying those to search for %INCLUDE file, 10-54
 - Disk storage,
 - RECORD input/output and, 12-2
 - sequential access files and, 12-9
 - STREAM input/output and, 11-24
 - DIVIDE built-in function, 14-41
 - Division,
 - by zero, ZERODIVIDE condition, 13-40
 - operator, 6-28
 - DO groups,
 - introduction, 4-15, 4-17
 - LEAVE statement, 10-24
 - named, 4-20
 - termination, 10-22
 - DO statement, (See also DO groups; Index variables)
 - in data item lists, 11-42
 - introduction, 4-15
 - REPEAT option, 10-16
 - DO statement (continued)
 - scope of condition prefix, 13-16
 - syntax, 10-4, 10-16
 - UNTIL option, 10-16
 - with IF statement, 10-21
 - DO WHILE statement,
 - introduction, 4-17
 - syntax, 10-5
 - with index variable, 10-13
 - DOT built-in function, 14-3, 14-42
 - Dummy arguments, 8-25
 - Dynamic storage area (DSA), 10-31
- E
- E,
 - in pictured-numeric specification, 5-53
 - specification of FLOAT BINARY constants, 5-13
 - specification of FLOAT DECIMAL constants, 5-9
 - E data format item,
 - input, specifications, 11-73
 - input, table, 11-75
 - output, introduction, 11-8
 - output, specifications, 11-55
 - output, table, 11-57
 - ECS (See Prime Extended Character Set)
 - EDIT option, (See also GET EDIT statement; PUT EDIT statement)
 - input, specifications, 11-76
 - output, specifications, 11-57
 - EDITOR,
 - ^ (caret) and, 6-4
 - PL/I and, 1-3
 - specifying ECS characters with, B-2

- ELSE option, 10-1
- EMPTY built-in function, 14-43
- Enabling conditions, 13-13
for debugging, 13-17
- End of file, 13-1
- END statement, (See also Blocks;
DO groups)
introduction, 4-3
multiple closure, 9-5
multiple closure, introduction,
4-21
syntax, 4-4
- End-of-file (See ENDFILE)
- ENDFILE condition, 12-32, 13-27
terminal input, 13-5
use of, 13-4
with STREAM input/output,
11-38
- ENDPAGE condition, 12-32, 13-28
and PRINT files, 11-33
with STREAM input/output,
11-38
- ENTRY data, (See also
Noncomputational data types)
constants, declaring, 7-31
constants, explicit
declaration, 9-14
internal representation, C-9
variables, 7-32, 8-42
variables, implementation,
10-43
variables, with DO loops,
10-21
- ENTRY option,
with EXTERNAL attribute, 8-34
- Entry points, 8-35
- ENTRY statement, 8-35
- EPFs (See Executable Program
Formats)
- ERF built-in function, 14-43
- ERFC built-in function, 14-44
- ERRLIST compiler option, 2-5
- ERROR condition, 13-29
raised for incomplete file
attributes, 12-29
risks of, 13-11
standard system action for
ENDFILE, 13-9
SYSTEM option, 13-12
use of, 13-9
- Error messages,
compiler, 2-2
suppressing with -NO_ERRITY,
2-6
suppressing with -SILENT, 2-11
- ERROR option, 5-79
- Errors (See Conditions)
- ERRITY compiler option, 2-6
- EVERY built-in function, 14-3,
14-45
- Exception handling (See
Conditions)
- Executable Program Formats, 3-1
- EXP built-in function, 14-46
- EXPLIST compiler option, 2-6
- Exponentiation,
operator, 6-30
- Expressions, (See also Aggregate
promotion; Data conversion;
Operators)
aggregates and, 6-43
built-in functions and, 6-8
introduction, 4-11, 6-2
operators, discussion, 6-25
- Extended Character Set (See
Prime Extended Character Set)

Extent expressions,
 defined, 7-1, 7-26
 variable, and REFER option,
 7-29
 variable, for parameters, 8-24
 variable, in RETURNS
 descriptors, 8-19
 variables in, 7-28

EXTERNAL attribute,
 FILE and ENTRY constants, 7-31
 in DECLARE statement, 8-34
 variables, 7-30

External procedures,
 EXTERNAL ENTRY declarations,
 8-33
 in other languages, interface,
 1-3
 introduction, 8-31
 PMA, and SHORICALL option,
 8-44
 variables, scope attributes,
 7-30

F

F,
 specification of FIXED BINARY
 constants, 5-10
 specification of FIXED DECIMAL
 constants, 5-5

F data format item,
 input, introduction, 11-20
 input, specifications, 11-76
 input, table, 11-77
 output, introduction, 11-8
 output, specifications, 11-58
 output, table, 11-59

F(n),
 in pictured-numeric
 specification, 5-41

-FH (See -FULL_HELP)

FILE,
 BIND command, 3-2

File attributes,
 CLOSE statement and, 12-31
 completing, 12-27
 conflicting, 12-29
 CONSTANT, 12-33
 DIRECT, 12-22
 implied in input/output
 statements, table, 12-27
 implied, table, 12-28
 INPUT, 11-27
 input/output statement
 requirements, 12-29
 input/output statement
 requirements, table, 12-30
 introduction, 12-20
 KEYED, 12-10, 12-22
 merging, 12-27
 OUTPUT, 11-27
 PRINT, 11-27, 11-33, 12-22
 required options, table, 12-30
 SEQUENTIAL, 12-22
 table, 12-21
 table of legal statements,
 12-22
UPDATE, 12-12
 VARIABLE, 12-33

File data,
 constants, with STREAM
 input/output, 11-25
 variables, introduction, 12-33
 variables, with FILE option,
 11-25

FILE data, (See also
 Noncomputational data types)
 constants, contextual
 declaration, 9-18
 constants, declaring, 7-31
 internal representation, C-10
 variables, 7-32

FILE option,
 default, 11-24
 input, specifications, 11-78
 output, specifications, 11-60
 with PUT or GET statements,
 11-24
 with READ or WRITE statements,
 11-35

- Files, (See also DAM files; Direct access files; File attributes; MIDASPLUS files; RECORD input/output; SAM files; Sequential access files; STREAM input/output) attribute merging and completion, 12-27 default title, 12-26 ENDFILE condition, 13-27 ENDPAGE condition, 13-28 errors in reading or writing, 13-31 implicit and explicit opening, 12-26 KEY condition, 13-31 NAME condition, 13-31 STREAM, description, 11-31 text, inserting in program, 10-53 TRANSMIT condition, 13-37 UNDEFINEDFILE condition, 13-38
- FINISH condition, 13-29
- FIXED BINARY data, assignments, table, 5-13 constants, table, 5-11 discussion, 5-10 internal representation, C-2 variables, table, 5-12
- FIXED built-in function, 6-42, 14-46
- FIXED data, (See also Scale factor of arithmetic data; Scale of arithmetic data) declaring, 4-23
- FIXED DECIMAL data, discussion, 5-5 internal representation, C-2
- FIXEDOVERFLOW condition, 13-30, 13-32 SIZE condition and, 13-33
- FLOAT BINARY data, constants, table, 5-14 discussion, 5-13 internal representation, C-3
- FLOAT built-in function, 6-42, 14-47
- FLOAT data, (See also Scale factor of arithmetic data; Scale of arithmetic data) declaring, 4-26
- FLOAT DECIMAL data, discussion, 5-9 internal representation, C-3
- Floating-point rounding (See -FRN)
- FLOOR built-in function, 14-48, 14-91 introduction, 4-30
- Flow of control, 10-1 introduction, 4-13
- FOFL (See FIXEDOVERFLOW)
- FOPT (See -FULL_OPTIMIZE)
- FORMAT data, (See also Noncomputational data types) constants, declaring, 7-31 constants, explicit declaration, 9-14 variables, 7-32, 11-45
- FORMAT data type, constants, 11-45
- Format items, A, 11-47, 11-66 B, 11-48, 11-67 C, 11-52, 11-70 classification, 11-44 COLUMN, 11-52, 11-71 data, matching data values to, 11-43 E, 11-55, 11-73 F, 11-58, 11-76 input control, list, 11-22 input data, list, 11-20 input data, table, 11-21 introduction, 11-6 LINE, 11-60 output control, list, 11-10 output data, list, 11-8 P, 11-62, 11-84

Format items (continued)

PAGE, 11-63
remote, 11-44, 11-64, 11-85
SKIP, 11-64, 11-85
TAB, 11-65
X, 11-66, 11-86

Format lists,

introduction, 11-6
repetition factors, 11-44b
variables and expressions in,
11-44c

FORMAT statement, 11-44

FORMS Management System, 1-7
PL/I interface to, H-1

-FORMS option, 12-26, H-1

FORTRAN programming language,
and two-dimensional arrays,
5-61

control codes, putting in
files, 12-25
PL/I interface to, 1-3

FREE statement, 7-7
syntax, 7-19

-FRN compiler option, 2-6

FROM option, 11-35, 12-15

-FTN option, 12-26

-FULL_HELP compiler option, 2-6

-FULL_OPTIMIZE compiler option,
2-7

Function procedures, (See also
Built-in functions)

arguments and parameters,
8-13, 8-21
differences from subroutine
procedures, 8-21
introduction, 4-49, 8-11
referencing, 8-12
returned values, location in
memory, D-1

Function procedures (continued)

returning aggregate values,
8-16
returning file values, 12-34
returning from, 8-13

-FUNIT option, 12-25

G

GENERIC attribute, 8-39

GET DATA statement,
errors, 13-31
introduction, 4-49, 11-17
NAME condition, 13-31
specifications, 11-72

GET EDIT statement, (See also
Format items)
control format items, list,
11-22
data format items, list, 11-20
data format items, table,
11-21
introduction, 11-19
specifications, 11-76

GET LIST statement,
flowchart, 11-81, 11-83
introduction, 4-4, 4-48, 11-15
specifications, 11-79
table, 11-82
with CHARACTER variables,
introduction, 4-35
with STRING option, 11-23

GET statement, 11-36
aggregate data items, 11-42
COPY option, 11-72
detailed specifications, 11-66
DO loops in data item lists,
11-42
establishing data items, 11-42
FILE option, 11-24, 11-78
files and devices, 11-24
introduction, 4-4, 11-13
SKIP option, 11-85
syntax, 11-13

- GET STRING statement,
 illegal options and format
 items, 11-24
 introduction, 11-23
 specifications, 11-85
 syntax, 11-23
- Glossary of PL/I terms, G-1
- GO TO statement, 10-22
 block invocation and, 10-40
 in on-units, 13-9
 introduction, 4-21
 on-units and, 13-4
 with LABEL expressions, 10-24
- GOTO (See GO TO)
- Groups (See Blocks; DO groups)
- H
- H (See -HELP)
- Hardware errors,
 data transmission, TRANSMIT
 condition, 13-37
- HBOUND built-in function, 8-29,
 14-3, 14-49
- HELP compiler option, 2-7
 (See also -FULL_HELP)
- HELP, BIND command, 3-3
- Hexadecimal notation,
 for BIT constants, 5-27
- HIGH built-in function, 14-3,
 14-50
- I
- I,
 in pictured-numeric
 specification, 5-52
 specification of COMPLEX
 constants, 5-15
- I (See -INPUT)
- IBM PL/I,
 features not supported in Prime
 PL/I, list, E-2
- Identifiers, (See also DECLARE
 statement)
 and noncomputational constants,
 7-31
 generic, 8-39
 introduction, 4-7
 resolving references, 9-27
- IF statement,
 DO statement and, 4-14, 10-21
 introduction, 4-13
 nested, 10-3
 scope of condition prefix,
 13-17
 syntax, 10-1
- IGNORE option, 12-7
- IMAG built-in function, 14-51
- IMAG pseudovvariable, 14-95
- Implementation-defined language
 features, 1-2
- IN option,
 syntax, 7-19
 with ALLOCATE statement, 7-15
- %INCLUDE statement, 10-53
 use with search rules facility,
 10-54
- INDEX built-in function, 14-51
 introduction, 4-38
- Index variables, (See also DO
 statement)
 introduction, 4-18
 multiple specifications, 10-19
 nonnumeric, 10-20
 numeric, 10-6
- Infix operators,
 * 6-27
 ** 6-30
 / 6-28
 &, |, and !, 6-35

Infix operators (continued)

- * , table, 6-27
- + and -, 6-25
- + and -, table, 6-26
- / , table, 6-29
- || or !!, 6-31
- comparison, 6-32

Inheriting variables, 10-31
(See also Scope rules)

INIT (See INITIAL)

INITIAL attribute, 7-26
arrays, 5-73
initializing variables, 7-2
scalars, 5-72
structures, 5-74
variables in, 5-75

INITIAL expressions,
variables in, 7-28

Initializing variables (See
INITIAL attribute)

-INPUT compiler option, 2-7

INPUT file attribute, 11-27

Input/output, (See also RECORD
input/output; STREAM
input/output; Terminal input)
devices, table, 12-25
introduction, 4-46
statements, required file
attributes, table, 12-30

Interactive program environment,
1-5

Interface to other languages,
1-3
(See also External procedures)
PMA, and SHORTCALL option,
8-44

INTERNAL attribute,
variables, 7-30

Internal buffers,
and RECORD input/output, 12-17

Internal procedures,
and NONQUICK option, 8-45
introduction, 8-4
placement in program, 8-8

INTO option, 11-35

iSUB defining, 5-70, 7-25

K

K,
in pictured-numeric
specification, 5-53

KEY condition, 12-32, 13-31

KEY option, 12-12

KEYED file attribute, 12-10,
12-22
KEY condition, 13-31

KEYED SEQUENTIAL files (See
MIDASPLUS files)

KEYFROM option,
KEY condition, 13-31
updating DAM files, 12-13
with MIDASPLUS files, 12-15

Keys,
in direct access files,
defined, 12-4

KEYTO option, 12-17

L

-L (See -LISTING)

LABEL data, (See also
Noncomputational data types;
Statement labels)
constants, arrays of, 9-15
constants, declaring, 7-31
constants, explicit
declaration, 9-14
internal representation, C-8

- LABEL data (continued)
 subscripted label prefixes,
 Prime restriction, E-2
 variables, 7-32
 variables, implementation,
 10-40
- IBOUND built-in function, 8-29,
 14-3, 14-52
- LCASE compiler option, 2-7
 (See also -UPCASE)
 identifiers and, 4-8
- LEAVE statement, 10-24
 termination of multiple loops,
 10-25
- LENGTH built-in function, 14-53
- Level numbers, 5-64, 9-7
 (See also Structures)
- LI (See LIBRARY)
- LIBRARY, BIND command, 3-2
- LIKE attribute, 5-71, 9-13
- Line editor (See EDITOR)
- LINE output option and control
 format item,
 control format item,
 introduction, 11-10
 option, and PRINT files, 11-33
 option, introduction, 11-2
 specifications, 11-60
- LINENO built-in function, 11-33,
 14-3, 14-54
- LINESIZE option, 11-27
 error checking, 12-29
- Linked lists,
 and BASED storage class, 7-10
 with DO loops, 10-21
- Linking programs, 3-1
 with SEG loader, I-1
- LIST option (See GET LIST
 statement; PUT LIST
 statement)
- List processing (See Linked
 lists)
- %LIST statement, 10-53
- LIST_SEARCH_RULES command, 10-55
- LISTING compiler option, 2-8
 new page, forcing, 10-54
 suppressing and restarting,
 10-54
- LO (See LOAD)
- LOAD, BIND command, 3-2
- Locate mode input/output, 12-17
 (See also LOCATE statement)
 and BASED storage, 12-18
 and POINTER data type, 12-18
 output, 12-18
 READ with SET option, 12-18
- LOCATE statement,
 and BASED structures, 12-19
 and locate mode output, 12-19
 RECORD condition, 13-33
 syntax, 12-20
- LOG built-in function, 14-54
- LOG10 built-in function, 14-55
- LOG2 built-in function, 14-56
- Logical expressions, (See also
 BIT data)
 introduction, 4-15
- Logical operators, 6-4
 ^ 6-37
 &, |, !, discussion, 6-35
 ^, table, 6-37
 table, 6-5
- Looping (See DO groups; DO
 statement)
- LOW built-in function, 14-3,
 14-56

LSR (See LIST_SEARCH_RULES
Command)

M

Magnetic tape,
table of devices, 12-25

-MAP compiler option, 2-8

MAP, BIND command, 3-3

-MAPWIDE compiler option, 2-8

Mathematical built-in functions,
ACOS, 14-15
ASIN, 14-19
ATAN, 14-20
ATAND, 14-22
ATANH, 14-22
classification and summary,
14-8
COMPLEX, 14-32
CONJG, 14-33
COS, 14-35
COSD, 14-35
COSH, 14-36
ERF, 14-43
ERFC, 14-44
EXP, 14-46
IMAG, 14-51
introduction, 4-32
LOG, 14-54
LOG10, 14-55
LOG2, 14-56
REAL, 14-71
SIN, 14-75
SIND, 14-76
SINH, 14-76
SQRT, 14-78
TAN, 14-83
TAND, 14-84
TANH, 14-85

Matrixes (See Arrays)

MAX built-in function, 14-4,
14-57
and data conversion, 6-16
introduction, 4-31

Memory dump,
use of SNAP option, 13-20

MIDASPLUS files, 1-6
basic statements, 12-15
DAM files, compared to, 12-14
DELETE statement and, 12-14,
12-16
FROM option, 12-15
introduction, 12-3
KEYFROM option, 12-15
keys, 12-14, 12-17
KEYTO option, 12-17
READ statement, 12-16
REWRITE statement, 12-16
WRITE statement, 12-16

MIN built-in function, 14-58
introduction, 4-31

Miscellaneous built-in functions,
classification and summary,
14-13

DATE, 14-36
LINENO, 14-54
PAGENO, 14-67
TIME, 14-85
UNSPEC, 14-92
VALID, 14-93

MOD built-in function, 4-31,
14-59

Mode of arithmetic data,
defined, 5-4
derived common, 6-17
derived common, table, 6-18

Modular programming, 8-1
(See also Blocks)

Multiple Index Data Access System
(See MIDASPLUS)

Multiplication operator, 6-27

MULTIPLY built-in function,
14-62

N

NAME condition, 12-32, 13-31

- Named constants, 7-31
- Names (See Identifiers)
- Naming programs, 2-1
- NAPRE (See
-ALLOW_PRECONNECTION)
- NB (See -BINARY)
- NBIG (See -BIG)
- NCOP (See -COPY)
- NDBG (See -DEBUG)
- NERRL (See -ERRLIST)
- NESTING compiler option, 2-9
- NEXP (See -EXPLIST)
- NFRN (See -FRN)
- NL (See -LISTING)
- NMA (See -MAP)
- NNE (See -NESTING)
- NO_STRINGSIZE option, 2-13
(See also NOSTRINGSIZE
condition)
- NOCONVERSION condition, 13-15
- NOFF (See -OFFSET)
- NOFIXEDOVERFLOW condition, 13-30
- %NOLIST statement, 10-53
- Noncomputational data types,
and data conversion, 6-38
introduction, 5-3
- NONE option, 5-79
- NONQUICK option, 8-45
- NOOVERFLOW condition, 13-32
- NOSIZE option, 12-25
- NOSTRINGSIZE condition, 13-36
(See also -NO_STRINGSIZE
option)
- Not,
logical operation, 6-37
- NOUNDERFLOW condition, 13-39
- NOVF (See -OVERFLOW)
- NOZERODIVIDE condition, 13-40
- NPROD (See -PRODUCTION)
- NRA (See -RANGE)
- NSOF (See -STORE_OWNER_FIELD)
- NSTAT (See -STATISTICS)
- NSTRZ (See -NO_STRINGSIZE)
- NULL built-in function, 14-63
and linked lists, 7-13
- Null string,
and BIT data, 5-26
and CHARACTER data, 5-22
introduction, 4-34
- Numeric data (See Arithmetic
data)
- NXREF (See -XREF)
- O
- Octal notation,
for BIT constants, 5-27
- OFFSET built-in function, 7-18,
14-3, 14-63
- OFFSET compiler option, 2-9
- OFFSET data, (See also
Noncomputational data types)
variables, 7-14
- OFL (See OVERFLOW)

- ON statement, 13-2
 - (See also Conditions; On-units)
 - REVERT statement and, 13-18
 - scope of condition prefix, 13-17
 - syntax, 13-2
 - SYSTEM option, 13-3, 13-12
 - with multiple condition names, 13-3
 - with sequential access files, 12-6

- On-units, (See also Conditions; ON statement)
 - abnormal termination, 13-4, 13-9
 - as blocks, 13-3
 - BEGIN/END block, 13-2
 - block invocation and termination, 10-27
 - defined, 13-2
 - implementation, 10-47
 - normal termination, 13-4
 - ON statement, 13-4
 - pseudovariabes in, 13-7
 - REVERT statement, 13-18
 - single statement, 13-2
 - termination, 13-4, 13-18

- ONCHAR built-in function, 14-64
 - and SIGNAL statement, 13-41
 - use in CONVERSION on-units, 13-6

- ONCHAR pseudovariabes, 13-7, 14-96
 - risks of, 13-9

- ONCODE built-in function, 14-64
 - use in on-units, 13-40
 - values, list, F-1

- ONCODE messages,
 - for ERROR condition, 13-9

- ONFIELD built-in function, 12-33, 14-65

- ONFILE built-in function, 12-33, 14-65

- ONKEY built-in function, 12-33, 14-66

- ONLOC built-in function, 14-66
 - and SIGNAL statement, 13-41

- ONSOURCE built-in function, 14-67
 - use in CONVERSION on-units, 13-6

- ONSOURCE pseudovariabes, 13-7, 14-96
 - risks of, 13-9

- OPEN statement,
 - default file title, 12-26
 - errors, 13-38
 - file attributes and, 12-20
 - options, list, 11-27
 - sequential access files, 12-6
 - STREAM files, 11-25
 - TITLE option, with RECORD files, 12-23
 - TITLE option, with STREAM files, 11-26
 - UNDEFINEDFILE condition, 13-38
 - UPDATE option, 12-12

- Operating environment, PRIMOS, 1-4

- Operating system (See PRIMOS)

- Operators, (See also Infix operators; Prefix operators)
 - and parentheses, 6-6
 - arithmetic, 6-3
 - arithmetic, introduction, 4-11
 - comparison, 6-3
 - comparison, introduction, 4-15
 - concatenation, 6-5
 - exponentiation, 6-3
 - expression, discussion, 6-25
 - infix, 6-3
 - logical, 6-4
 - prefix, 6-3
 - priority, 6-6
 - priority, introduction, 4-11
 - priority, table, 6-7

- OPTIMIZE compiler option, 2-9
 - and NONQUICK procedures, 8-45

Options, (See also Compiler options)

-ANSI, 11-29
 -APPEND, 11-29, 12-9, 12-24
 BY, 10-6
 BY NAME, 5-65
 COPY, 11-14, 11-72
 -CTLASA, 12-25
 -DAM, 12-24
 DATA, input, 11-17, 11-72
 DATA, output, 11-4, 11-53
 -DEVICE, 11-29, 12-24
 EDIT, input, 11-19, 11-21, 11-76
 EDIT, output, 11-6, 11-9, 11-57
 ELSE, 10-1
 ENTRY, 8-34
 ERROR, with DEFAULT statement, 5-79
 FILE, 11-24, 11-35, 11-60, 11-78
 file, required attributes, table, 12-30
 -FORMS, 12-26, H-1
 FROM, 11-35, 12-15
 -FIN, 12-26
 -FUNIT, 12-25
 IGNORE, 12-7
 IN, 7-15
 INTO, 11-35
 KEY, 12-12
 KEYFROM, 12-13, 12-15
 KEYTO, 12-17
 LINE, 11-2, 11-33, 11-60
 LINESIZE, 11-27, 12-29
 LIST, input, 11-15, 11-79
 LIST, output, 11-4, 11-61
 NONE, 5-79
 NONQUICK, 8-45
 -NOSIZE, 12-25
 OPTIONS(MAIN), introduction, 4-3
 OTHERWISE, 10-25
 PAGE, 11-2, 11-33, 11-40, 11-63
 PAGE, output, introduction, 4-48
 PAGESIZE, 11-27, 12-29
 POINTER OPTIONS(SHORT), 7-32
 -RECL, 12-25
 RECURSIVE, 8-39
 REFER, and BASED storage, 7-29
 REPEAT, 10-16

Options (continued)

RETURNS, 8-12, 8-35
 -SAM, 12-23
 SET, 7-8, 12-18
 SHORCALL, 8-44
 SKIP, input, 11-14, 11-85
 SKIP, output, 11-2, 11-64
 SKIP, output, introduction, 4-47
 STRING, input, 11-23, 11-85
 STRING, output, 11-12, 11-64
 SYSTEM, 5-79, 10-49
 TAB, 11-28, 12-29
 THEN, 10-1
 TITLE, with RECORD files, 12-20, 12-23
 TITLE, with STREAM files, 11-26, 11-28
 TO, 10-6
 UNTIL, 10-16
 WHEN, 8-40, 10-25

OPTIONS(MAIN) option, introduction, 4-3

Or, logical operation, 6-35

OTHERWISE option, 10-25

Output (See RECORD input/output; STREAM input/output)

OUTPUT file attribute, 11-27

-OVERFLOW compiler option, 2-10 and FIXEDOVERFLOW condition, 13-30

OVERFLOW condition, 13-32 (See also FIXEDOVERFLOW condition)

Overlaying storage, BASED storage, 7-21 DEFINED attribute and, 7-24 introduction, 7-20 machine independent, 7-23 simple overlaying, 7-23 string overlaying, 7-23

P

P,
 specifying system defaults for constants, 5-79

P data format item,
 input, specifications, 11-84
 input, table, 11-84
 output, introduction, 11-8
 output, specifications, 11-62
 output, table, 11-62

Page headings,
 printing, 11-38

PAGE option, introduction, 4-48

PAGE output option and control format item,
 control format item,
 introduction, 11-10
 option, and PRINT files, 11-33
 option, introduction, 11-2
 specifications, 11-63
 with ENDPAGE on-unit, 11-40

%PAGE statement, 10-53

PAGENO built-in function, 11-33,
 14-3, 14-67

PAGENO pseudovvariable, 11-34,
 14-96

PAGESIZE option, 11-28
 error checking, 12-29

Paper tape,
 table of devices, 12-25

PARAMETER storage type,
 declaration, 9-14
 defined, 7-3

Parameters,
 arguments, relation to, 8-21
 arrays, 8-29
 dummy arguments and, 8-25
 for function procedures, 8-13
 for subroutine procedures, 8-9
 scope rules, 8-10
 variable extent expressions in,
 8-24

Parentheses,
 in expressions, 6-6

Period,
 in pictured-numeric specifications, 5-38

Phantom user program environment,
 1-5

PICTURE data, (See also
 Computational data types;
 Pictured-numeric data)
 ALIGNED attribute and, 5-67
 assignment errors, 13-36
 conversion, 6-41
 defined, 5-3
 DEFINED attribute and, 5-70
 internal representation, C-6
 introduction, 5-29
 pictured-string, discussion,
 5-29
 repetition factors, 5-30

Picture-specification characters,
 \$ 5-50
 * 5-42
 + 5-35, 5-46
 , 5-38
 - 5-35, 5-46
 . 5-38
 / 5-38
 9, 5-30, 5-32
 A, 5-30
 B, 5-38
 CR, 5-35
 DB, 5-35
 E, 5-53
 F(n), 5-39
 I, 5-52
 K, 5-53
 R, 5-52
 S, 5-32, 5-46
 T, 5-52
 V, 5-39, 5-45
 X, 5-30
 Y, 5-42
 Z, 5-42

Pictured-numeric data,
 assigning values, table, 5-34
 COMPLEX pictures, 5-56
 discussion, 5-31
 drifting signs, 5-46

- Pictured-numeric data (continued)
 drifting signs, table, 5-48
 FLOAT symbols, 5-53
 FLOAT symbols, table, 5-54
 insertion character symbols,
 table, 5-38
 insertion characters, 5-38
 internal representation, 5-33
 noninteger values, 5-39
 overpunched sign symbols, 5-52
 overpunched signs, table,
 5-52, 5-53
 representing signs, 5-35
 scale factor symbols, 5-39
 scale-factor symbols, table,
 5-40
 signed values, table, 5-37
 static and drifting \$, 5-50
 static and drifting \$, table,
 5-51
 V and zero suppression, 5-45
 V and zero suppression, table,
 5-47
 zero suppression, 5-42
 zero suppression, table, 5-43
- PL/I Subset G,
 and Prime PL/I, 1-1
- PLI command, 2-2
- PMA (Prime Macro Assembler),
 PL/I interface to, 1-3, 8-44
- POINTER built-in function, 7-16,
 14-3, 14-68
- POINTER data, (See also
 Noncomputational data types)
 internal representation, C-8
 linked lists and, 7-10
 locate mode input/output and,
 12-18
 SHORT option, 7-32
 variables, contextual
 declaration, 9-19
 variables, defined, 7-3
 variables, referencing with ->
 operator, 7-9
- POINTER OPTIONS(SHORT) data,
 7-32
 internal representation, C-8
- POSITION attribute, 5-69
- PREC (See PRECISION)
- PRECISION built-in function,
 14-68
- Precision of arithmetic data,
 arguments to built-in
 functions, 14-5
 conversion, for built-in
 functions, 14-4
 defined, 5-4
 in data conversion, 6-21
 introduction, 4-23
- Prefix operators,
 ^ 6-37
 + and -, 6-36
 + and -, table, 6-37
- Preopened file units (See
 -ALLOW_PRECONNECTION)
- Prime ECS (See Prime Extended
 Character Set)
- Prime Extended Character Set,
 B-1
 table of, B-7
- Prime extensions to PL/I,
 A input data format item with
 no specification, 11-21
 BYTE built-in function, 14-29
 %INCLUDE statement, 10-53
 LEAVE statement, 10-24
 %LIST, 10-53
 listed, E-1
 %NOLIST, 10-53
 %PAGE statement, 10-53
 RANK built-in function, 14-70
 READ and WRITE with STREAM
 input/output, 11-31, 11-34,
 11-35
 %REPLACE statement, 10-53
 SELECT statement, 10-25
 SIZE built-in function, 14-77
 UNTIL option of DO statement,
 10-16
- Prime restrictions to PL/I, 1-2
 FIXEDOVERFLOW detection, 13-30

PL/I Reference Guide

Prime utilities,
FORMS Management System, 1-7
MIDASPLUS, 1-6
Source Level Debugger, 1-7

PRIMOS,
and PL/I, 1-4
environments, 1-5

PRIMOS commands,
BIND, 3-1
PL1, 2-2
RESUME, 3-3
SPOOL, -FTN option, 12-26

PRINT file attribute, 11-27,
11-33, 12-22

PRINT files,
ENDPAGE condition, 13-28

Printers,
and RECORD output, 12-2
table of devices, 12-25

PROCEDURE statement,
block invocation and
termination, 10-27
introduction, 4-3
RETURNS option, 8-12
scope of condition prefix,
13-16

Procedures, (See also Blocks;
External procedures; Function
procedures; Internal
procedures; Scope rules;
Subroutine procedures)
entry points, 8-35
external, introduction, 4-49,
8-31
generic entry names, 8-39
internal, introduction, 4-49
introduction, 8-1
recursive, 8-37
SHORCALL and NONQUICK, 8-44
subprocedures, dropping into,
8-7
subroutine and function,
differences, 8-21
summary of rules, 8-45

PROD built-in function, 14-3,
14-69

-PRODUCTION compiler option,
2-10

Program blocks (See Blocks)

Program environments,
batch job, 1-6
interactive, 1-5
phantom user, 1-5

Program names, 2-1
introduction, 4-3

Pseudovariables,
defined, 14-1
IMAG, 14-95
ONCHAR, 13-7, 14-96
ONSOURCE, 13-7, 14-96
PAGENO, 14-96
REAL, 14-95
STRING, 14-96
SUBSTR, 13-35, 14-97
UNSPEC, 14-98
use in on-units, 13-7
use of, 14-95

Punched cards,
and GET EDIT, 11-19
RECORD input/output, 12-2,
12-8
STREAM input/output, 11-31

PUT DATA statement,
introduction, 4-49, 11-4
specifications, 11-53
structures and, 11-54
use in debugging, 11-5

PUT EDIT statement, (See also
Format items; Format lists)
control format items,
introduction, 11-9
data format items, table, 11-9
introduction, 4-48, 11-6
repetition factors, 11-12
specifications, 11-57
STRINGSIZE condition, 13-36

PUT LIST statement,
introduction, 4-6, 4-46, 11-4
specifications, 11-61

- PUT statement, 11-36
 aggregate data items, 11-42
 detailed specifications, 11-46
 DO loops in data item lists, 11-42
 establishing data items, 11-42
 FILE option, 11-24, 11-60
 files and devices, 11-24
 introduction, 4-6, 11-2
 LINE option, 11-60
 PAGE option, 11-40, 11-63
 raising ENDPAGE condition, 13-28
 SKIP option, 4-47, 11-64
 syntax, 11-2
- PUT STRING statement,
 illegal options and format items, 11-13
 introduction, 11-12
 specifications, 11-64
 syntax, 11-13
- Q
- Qualified name,
 defined, 5-63
- Quartal notation,
 for BIT constants, 5-27
- QUIT, BIND command, 3-3
- R
- R, (See also Remote format item)
 in pictured-numeric
 specification, 5-52
- RANGE compiler option, 2-11
- RANGE keyword,
 variable-name test in DEFAULT
 statement, 5-77
- RANK built-in function, 14-70
- READ statement,
 IGNORE option, 12-7
 RECORD condition, 13-33
- READ statement (continued)
 SET option, 12-18
 with DAM files, 12-11
 with MIDASPLUS files, 12-16
 with sequential access files, 12-6
 with STREAM input/output, 11-31, 11-34, 11-35
- REAL attribute (See Mode of arithmetic data)
- REAL built-in function, 14-71
- REAL pseudovalue, 14-95
- RECL option, 12-25
- RECORD condition, 12-32, 13-33
- RECORD input/output, (See also DAM files; Direct access files; Files; MIDASPLUS files; SAM files; Sequential access files; STREAM input/output)
 data types, 12-8
 introduction, 4-49, 12-1
 keys, introduction, 12-4
 locate mode, 12-17
 noncharacter records, 12-8
 punched cards, 12-2
 STREAM input/output and, 12-1
- RECURSIVE option, 8-39
- Recursive procedures, 8-37
 AUTOMATIC storage and, 7-6
 block structure, 10-34
- REFER option,
 BASED storage and, 7-29
- Remote format item,
 FORMAT variables and, 11-45
 input, specifications, 11-85
 introduction, 11-44a
 output, specifications, 11-64
- REPEAT option, 10-16
- Repetition factors,
 in CHARACTER constants, 5-23
 in format lists, 11-44b

Repetition factors (continued)
in format lists, variables and
expressions in, 11-44c
in INITIAL values, 7-26
in pictured-numeric
specifications, 5-34
in pictured-string
specifications, 5-30
in PUT EDIT statement, 11-12

%REPLACE statement, 10-53

Replication factors,
and repetition factors, in
INITIAL values, 7-27

Reserved words,
absence of, 4-8

RESUME command, 3-3

RETURN statement,
FINISH condition, 13-29
for function procedures, 8-14
for subroutine procedures, 8-6

RETURNS option,
EXTERNAL attribute and, 8-35
function procedures, 8-12
syntax, 8-14
variable extent expressions,
8-19

REVERSE built-in function, 14-71

REVERT statement, 13-18
implementation, 10-50
syntax, 13-19

REWRITE statement,
RECORD condition, 13-33
updating DAM files, 12-13
with MIDASPLUS files, 12-16

ROUND built-in function, 14-72

Runfiles, creating, 3-1

Running programs, 3-3

S

S,
in pictured-numeric
specification, 5-32, 5-46

SAM files (See Sequential access
files)

-SAM option, 12-23

Scalar data,
defined, 4-42

Scale factor of arithmetic data,
defined, 5-5
in data conversion, 6-21
introduction, 4-25

Scale of arithmetic data,
defined, 5-4
derived common, 6-14
derived common, table, 6-15
introduction, 4-23

Scope rules, 9-21
contextual and implicit
declarations, 9-26
explicit declarations, 9-25
for condition prefixes, 13-16
for parameters, 8-10
for procedures, 8-8
for variables, 7-30

search rules facility,
use of with %INCLUDE
statements, 10-54

SEG loading utility, I-1

Segment-spanning code (See -BIG)

SELECT statement, 10-25

Separators,
for STREAM input, 11-16

Sequential access files, (See
also Direct access files; SAM
files)
-APPEND option, 12-9
appending to, 12-9
basic statements, 12-5
CLOSE statement, 12-7

- Sequential access files
(continued)
 - disk files, 12-9
 - example, 12-5
 - IGNORE option, 12-7
 - introduction, 12-2
 - ON statement, 12-6
 - OPEN statement, 12-6
 - READ statement, 12-6
 - SAM files, introduction, 12-3
 - WRITE statement, 12-7
- SEQUENTIAL file attribute, 12-22
- SET option, 7-8, 12-18
 - syntax, 7-19
- SET_SEARCH_RULES command, 10-55
- SHORT option, 7-32
- SHORTCALL option, 8-44
- SIGN built-in function, 14-74
- SIGNAL statement,
 - CONDITION condition and, 13-26
 - condition-handling built-in functions and, 13-41
 - ONCHAR built-in function and, 13-41
 - ONLOC built-in function and, 13-41
 - syntax, 13-19
 - uses of, 13-20
- SILENT compiler option, 2-11
- SIN built-in function, 14-75
- SIND built-in function, 14-76
- SINH built-in function, 14-76
- SIZE built-in function, 14-77
- SIZE condition, 13-33
 - and FIXEDOVERFLOW, 13-33
 - errors in FIXED DECIMAL assignments, 5-7
- SKIP option and control format item,
 - SKIP option and control format item (continued)
 - input control format item, introduction, 11-22
 - input option, introduction, 11-14
 - input, specifications, 11-85
 - output, 4-47, 11-3
 - output control format item, introduction, 11-10
 - output option, introduction, 11-2
 - output, specifications, 11-64
- SNAP option, 13-20
- SOF (See -STORE_OWNER_FIELD)
- SOME built-in function, 14-3, 14-78
- SOURCE compiler option, 2-11
- Source Level Debugger, 1-7
- Source listing (See -LISTING)
- SPACE compiler option, 2-11
(See also -TIME)
- Spacing,
 - in statements, introduction, 4-10
- SPOOL command,
 - FIN option, 12-26
- SQRT built-in function, 14-78
- SSR (See SET_SEARCH_RULES Command)
- Stack frame,
 - format, D-2
 - obtaining information with SNAP option, 13-21
- Statement labels,
 - as named constants, 7-31, 9-14
 - block containment, 9-23
 - introduction, 4-21
 - used in FORMAT statement, 11-44a

Statements,

ALLOCATE, 7-7
 assignment, 4-5
 assignment, introduction, 4-2
 blocks, 9-1
 CALL, 8-3
 classification, 4-2
 CLOSE, 12-7, 12-31
 DECLARE, for files, 11-25,
 12-20
 DECLARE, introduction, 4-23,
 9-6
 DEFAULT, 5-75
 defined, 4-2
 DELETE, 12-14, 12-16
 DO, 10-4
 DO WHILE, 10-5, 10-13
 DO, groups, 4-17
 DO, in data item lists, 11-42
 DO, introduction, 4-15
 END, introduction, 4-3
 ENTRY, 8-35
 FORMAT, 11-44
 FREE, 7-7
 GET, 11-13, 11-24, 11-36,
 11-66
 GET DATA, 4-49, 11-17, 11-72
 GET EDIT, 11-19, 11-21, 11-76
 GET LIST, 11-15, 11-79, 11-81
 GET LIST, introduction, 4-4,
 4-48
 GET STRING, 11-23, 11-85
 GET, introduction, 4-4
 GO TO, 4-21, 10-22
 groups, 9-1
 IF, 4-13, 10-1, 10-21
 %INCLUDE, 10-53
 input/output, file attribute
 requirements, 12-29
 input/output, file attribute
 requirements, table, 12-30
 keyword, 4-2
 LEAVE, 10-24
 %LIST, 10-53
 LOCATE, 12-19
 maximum length, 4-2
 %NOLIST, 10-53
 ON, 13-2
 ON, with sequential access
 files, 12-6
 OPEN, and file attributes,
 12-20
 OPEN, TITLE option, 12-23
 OPEN, UPDATE option, 12-12

Statements (continued)

OPEN, with sequential access
 files, 12-6
 OPEN, with STREAM files, 11-25
 %PAGE, 10-53
 PROCEDURE, introduction, 4-3
 PUT, 11-2, 11-24, 11-36, 11-46
 PUT DATA, 4-49, 11-4, 11-53
 PUT EDIT, 11-6, 11-9, 11-57
 PUT EDIT, introduction, 4-48
 PUT LIST, 11-4, 11-61
 PUT LIST, introduction, 4-6,
 4-46
 PUT STRING, 11-12, 11-64
 PUT, introduction, 4-6
 READ, IGNORE option, 12-7
 READ, SET option, 12-18
 READ, with DAM files, 12-11
 READ, with MIDASPLUS files,
 12-16
 READ, with sequential access
 files, 12-6
 READ, with STREAM input/output,
 11-34 to 11-36
 %REPLACE, 10-53
 RETURN, 8-6, 8-14
 REVERT, 10-50
 REWRITE, 12-13, 12-16
 SELECT, 10-25
 WRITE, 12-7, 12-13, 12-16
 WRITE, with STREAM
 input/output, 11-34 to 11-36
 STATIC storage class, 7-4
 defined, 7-3
 EXTERNAL attribute and, 7-30
 -STATISTICS compiler option,
 2-12
 STOP statement,
 FINISH condition, 13-29
 Storage,
 alignment of data, 5-66
 allocation errors, 13-34
 allocation, defined, 7-2
 AUTOMATIC, 7-3, 7-4
 BASED, 7-3, 7-8
 class and type, defined, 7-4
 classification, 7-2
 CONTROLLED, 7-3, 7-6
 DEFINED attribute, 7-3
 DEFINED attribute and, 5-67

- Storage (continued)
 - for block invocations, 10-28
 - formats, C-1
 - freeing, defined, 7-2
 - introduction, 7-1
 - overlying, 7-20
 - PARAMETER, 7-3, 9-14
 - POINTER variables, 7-3, 7-32
 - restrictions on programs, 1-2
 - STATIC, 7-3, 7-4
 - STORAGE condition and, 13-34
 - temporary, defined, 7-4
- STORAGE condition, 13-34
- Storage-handling built-in functions,
 - ADDR, 7-22, 14-3, 14-16
 - ALLOCATION, 7-7, 14-3, 14-18
 - classification and summary, 14-12
 - EMPTY, 14-43
 - NULL, 7-13, 14-63
 - OFFSET, 7-18, 14-3, 14-63
 - POINTER, 7-16, 14-3, 14-68
 - SIZE, 14-77
- STORE_OWNER_FIELD compiler option, 2-12
- STREAM input/output, (See also RECORD input/output)
 - description, 11-31
 - device independence, 11-30
 - establishing data items, 11-42
 - file information pointers and values, 11-34
 - formatting input, 11-19
 - formatting output, 11-6
 - input files, preparing, 11-15
 - introduction, 11-1
 - PRINT files, 11-33
 - program portability, 11-30
 - PUT and GET to files and devices, 11-24
 - READ/WRITE vs GET/PUT (table), 11-36
 - RECORD input/output and, 12-1
 - specifications, introduction, 11-30
 - using nonstandard READ and WRITE with, 11-31, 11-34, 11-35
- STRG (See STRINGRANGE)
- STRING built-in function, 14-3, 14-79
- String data, (See also BIT data; CHARACTER data; Computational data types)
 - assignment errors, 13-36
 - conversion, 6-24
 - conversion, table, 6-24
 - defined, 5-3
 - initializing, 5-74
 - introduction, 5-18
 - STRINGSIZE condition, 13-36
- STRING option, (See also GET STRING statement; PUT STRING statement)
 - input, specifications, 11-85
 - output, specifications, 11-64
- STRING pseudovalue, 14-96
- String-handling built-in functions,
 - AFTER, 14-17
 - BEFORE, 14-23
 - BIT, 14-26
 - BOOL, 14-27
 - BYTE, 14-29
 - CHARACTER, 14-31
 - classification and summary, 14-9
 - COLLATE, 14-32
 - COPY, 14-34
 - DECAT, 14-37
 - EVERY, 14-45
 - HIGH, 14-50
 - INDEX, 14-51
 - introduction, 4-37
 - LENGTH, 14-53
 - LOW, 14-56
 - RANK, 14-70
 - REVERSE, 14-71
 - SOME, 14-78
 - STRING, 14-79
 - SUBSTR, 14-80
 - TRANSLATE, 14-86
 - TRIM, 14-89
 - VERIFY, 14-94
- STRINGRANGE condition, 13-35

- STRINGSIZE compiler option, 2-13
- STRINGSIZE condition, 13-36
- Structures, (See also Aggregate promotion)
 - aggregate promotion, 6-45
 - arrays of, 5-65
 - BY NAME option, 5-65
 - card images and, 12-8
 - declaring, 9-7
 - DEFINED attribute and, 5-70
 - discussion, 5-63
 - in expressions, 6-44
 - initializing, 5-74
 - interleaved subscripts, 9-28
 - introduction, 4-44
 - level numbers, 9-7
 - LIKE attribute and, 5-71
 - operations on, 6-44
 - PUT DATA and, 11-54
 - qualified names, 5-63, 9-28
 - resolving references, 9-28
 - returning, from functions, 8-17
- STRZ (See STRINGSIZE)
- STRZ (See -STRINGSIZE)
- SUBRG (See SUBSCRIPTRANGE)
- Subroutine procedures,
 - arguments and parameters, 8-21
 - calling, 8-5
 - differences from function procedures, 8-21
 - introduction, 4-49, 8-3
 - parameters, 8-9
 - point of invocation, 8-6
 - returning from, 8-6
- Subscript errors, 13-13
 - SUBSCRIPTRANGE condition, 13-37
- SUBSCRIPTRANGE condition, 13-37
 - enabling, 13-13
- Subscripts (See Array subscripts)
- Substitutions,
 - in program text, 10-54
- SUBSTR built-in function, 14-80
 - introduction, 4-37
 - STRINGRANGE condition, 13-35
- SUBSTR pseudovariable, 14-97
 - STRINGRANGE condition, 13-35
- Substructures, 5-64
- SUBTRACT built-in function, 14-81
- Subtraction operator, 6-25
- SUM built-in function, 14-3, 14-82
- SYSIN, (See also Terminal input)
 - contextual declaration, 9-18
 - default FILE option, 11-24
 - terminal input file, 11-14
- SYSPRINT,
 - and PRINT file attribute, 11-33
 - contextual declaration, 9-18
 - default FILE option, 11-24
 - terminal output file, 11-2
- SYSTEM option, 5-79, 13-3
 - implementation, 10-49
 - with ERROR condition, 13-11
- T
- T,
 - in pictured-numeric specification, 5-52
- TAB output option and control
 - format item,
 - format item, specifications, 11-65
 - option, error checking, 12-29
 - option, with OPEN statement, 11-28
- TAN built-in function, 14-83

- TAND built-in function, 14-84
- TANH built-in function, 14-85
- Tape storage,
and RECORD input/output, 12-2
and STREAM input/output, 11-24
table of devices, 12-25
- Temporary variables,
introduction, 7-4
- Terminal input, (See also SYSIN)
entering data values, 11-16
introduction, 4-4
use of ON ENDFILE, 13-5
- Terminal output (See SYSPRINT)
- Terminals,
table of devices, 12-25
- Termination of program,
and FINISH condition, 13-29
- Text files,
inserting in program text,
10-53
- THEN option, 10-1
- TIME built-in function, 14-85
- TIME compiler option, 2-13
(See also -SPACE)
- TITLE option,
-ANSI option, with STREAM
files, 11-29
-APPEND option, with RECORD
files, 12-9, 12-24
-APPEND option, with STREAM
files, 11-29
-CTLASA option, 12-25
-DAM option, 12-24
-DEVICE option, with RECORD
files, 12-24
-DEVICE option, with STREAM
files, 11-29
errors, 13-38
file attributes and, 12-20
formats, 12-23
-FORMS option, 12-26
-FUNIT option, 12-25
- TITLE option (continued)
-NOSIZE option, 12-25
-RECL option, 12-25
-SAM option, 12-23
STREAM files and, 11-26, 11-28
- TO option, 10-6
- Top-down programming, 8-1
- TRANSLATE built-in function,
14-86
- TRANSMIT condition, 12-32, 13-37
raising with SIGNAL statement,
13-20
- TRIM built-in function, 14-89
- TRUNC built-in function, 4-30,
14-90
- Truncation,
of BIT data, 5-24
of CHARACTER data, 5-20
of FIXED DECIMAL variables,
5-7
- U
- UFL (See UNDERFLOW)
- UNALIGNED attribute, 5-66
- Unary plus and minus (See Prefix
operators)
- UNDEFINEDFILE condition, 12-32,
13-38
- UNDERFLOW condition, 13-39
- UNDF (See UNDEFINEDFILE)
- UNSPEC built-in function, 14-3,
14-92
- UNSPEC pseudovvariable, 14-98
- UNTIL option, 10-16

PL/I Reference Guide

-UPCASE compiler option, 2-13
(See also -LCASE)

UPDATE file attribute,
REWRITE statement and, 12-13
with DAM files, 12-12
WRITE statement and, 12-13

User-defined functions (See
Function procedures)

Utilities (See Prime utilities)

V

V,
in pictured-numeric
specification, 5-39, 5-45

VALID built-in function, 14-93

VARIABLE file attribute, 12-33

Variable-length lines,
processing unformatted, 11-34

Variables,
data types, introduction, 5-2
ENTRY, 7-32
FILE, 7-32
FORMAT, 7-32
in extent and INITIAL
expressions, 7-28
initializing, 5-72
LABEL, 7-32
scope, 7-30
temporary, 7-4

VERIFY built-in function, 14-94

W

WHEN option,
with GENERIC attribute, 8-40
with SELECT statement, 10-25

WHILE option (See DO WHILE
statement)

WRITE statement,
KEYFROM option, 12-13
RECORD condition, 13-33
updating DAM files, 12-13
with MIDASPLUS files, 12-16
with sequential access files,
12-7
with STREAM input/output,
11-31, 11-34, 11-35

X

X,
in pictured-string
specification, 5-30

X control format item,
input, introduction, 11-22
input, specifications, 11-86
output, introduction, 11-10
output, specifications, 11-66

-XREF compiler option, 2-13

Y

Y,
in pictured-numeric
specification, 5-45

Z

Z,
in pictured-numeric
specification, 5-42

ZDIV (See ZERODIVIDE)

ZERODIVIDE condition, 13-40

SURVEY



READER RESPONSE FORM

DOC5041-1LA PL/I Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

excellent very good good fair poor

2. Please rate the document in the following areas:

Readability: hard to understand average very clear

Technical level: too simple about right too technical

Technical accuracy: poor average very good

Examples: too many about right too few

Illustrations: too many about right too few

3. What features did you find most useful? _____

4. What faults or errors gave you problems? _____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

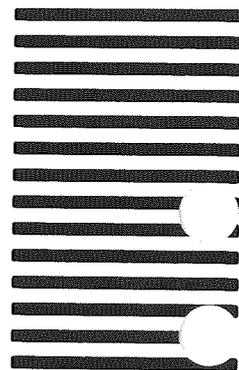
Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



READER RESPONSE FORM

DOC5041-1LA PL/I Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

excellent very good good fair poor

2. Please rate the document in the following areas:

Readability: hard to understand average very clear

Technical level: too simple about right too technical

Technical accuracy: poor average very good

Examples: too many about right too few

Illustrations: too many about right too few

3. What features did you find most useful? _____

4. What faults or errors gave you problems? _____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760



READER RESPONSE FORM

DOC5041-1LA PL/I Reference Guide

Your feedback will help us continue to improve the quality, accuracy, and organization of our user publications.

1. How do you rate the document for overall usefulness?

excellent very good good fair poor

2. Please rate the document in the following areas:

Readability: hard to understand average very clear

Technical level: too simple about right too technical

Technical accuracy: poor average very good

Examples: too many about right too few

Illustrations: too many about right too few

3. What features did you find most useful? _____

4. What faults or errors gave you problems? _____

Name: _____ Position: _____

Company: _____

Address: _____

_____ Zip: _____



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

PRIME

Attention: Technical Publications
Bldg 10B
Prime Park, Natick, Ma. 01760

