THE PL/1 SUBSET G REFERENCE GUIDE

PRIME Computer

IDR 4031

THE PL/I SUBSET G REFERENCE GUIDE IDR4031

This guide documents the operation of the Prime Computer and its supporting systems and utilities as implemented at Master Disk Revision Level 17 (Rev. 17).

PRIME

PRIME Computer, Inc.
500 Old Connecticut Path
Framingham, Massachusetts 01701

ACKNOWLEDGEMENTS

We wish to thank the members of the documentation team and also the non-team members, both customer and Prime, who contributed to and reviewed this book.

First Printing April 1980

All correspondence on suggested changes to this document should be directed to:

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

CONTENTS

APPENDICES

Part I
Introduction

SECTION 1

INTRODUCTION

DEFINITIONS

There are several versions of PL/I. The following names are used for the versions discussed in this guide.

PL/I: A general-purpose high-level programming language, defined in the American National Standards Institute (ANSI) publication "ANSI X3.53-1976".

PL/I Subset G: A subset of the full PL/I language. Subset G is defined in the draft proposed ANSI standard "BSR X3.74" Revision 8. For convenience, BSR X3.74 will hereafter be referred to as the ANSI standard for PL/I Subset G, although the process of official adoption is not yet complete as of Rev. 17.2.

PL1G: Prime's extended version of PL/I Subset G. The PL1G language conforms fully to BSR X3.74, except as noted below under Excluded Features of PL1G.

Certain PL/I-specific terms used in this introduction are formally defined in the glossary in Appendix A.

THIS DOCUMENT

This document is a programmer's guide to the PL/I Subset G language as implemented on the Prime system. The reader is expected to be familiar with some high-level language, and with programming in general, but not necessarily with PL/I or Prime computers. Users who need additional background in programming techniques or PL/I should consult an appropriate textbook. Some examples are:

Conway, Richard, and Gries, David, An Introduction to Programming - A Structured Approach Using PL/I and PL/C, Winthrop Publishers, Inc.

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

Pollack, Seymour V., and Sterling, Theodore C., A Guide to PL/I, Hall, Rinehart, and Winston.

This document contains the following:

- A general introduction.
- An overview of the PL1G language.
- All the information from ANSI X3.53-1976 and BSR X3.74 which a programmer needs to program in PL1G. Various details elaborated in the standards for the sake of completeness, but unlikely ever to be required in practice, have been omitted to limit this guide to a reasonable size.
- Complete information on all extensions to and implementation-defined features of Prime's PL/I Subset G.
- Complete information on the use of the PL1G compiler.
- A glossary of PL/I terms.
- A detailed comparison of PL/I and PL/I Subset G.
- Supplementary information useful in a variety of commonly encountered programming situations.

RELATED DOCUMENTS

Nearly all the information in The PL/I Subset G Reference Guide (this guide) relates directly to the PL1G language. Little general information on the Prime computer system is presented here. The following documents contain the additional information needed to program in PL1G on the Prime system.

The New User's Guide to Editor and Runoff

The PRIMOS editor is an interactive text-editing utility. It is used to enter new text into the computer, and to modify material previously entered. New programs not resident on media such as cards or tape are usually input to the system at a terminal using the editor.

The New User's Guide to Editor and Runoff contains a complete description of the editor. It also provides a basic introduction to the Prime system for those with little or no computer experience, and describes Runoff, Prime's text-formatting utility.

The Prime User's Guide

Complete instructions for creating, loading and executing programs in PL1G or any Prime language, plus extensive additional information on Prime system utilities for programmers, is found in The Prime User's Guide. The user's guide and this reference guide are complementary documents: both are essential to the PL1G programmer.

The user's guide also contains a complete description of all Prime documents.

The LOAD and SEG Reference Guide

Ordinary loading and execution of programs requires only the information given in The Prime User's Guide. Those who wish to control the load process in more detail, or otherwise take advantage of the full range of Prime loader capabilities, are referred to The LOAD and SEG Reference Guide.

The PRIMOS Subroutines Reference Guide

Prime offers a large selection of applications-level subroutines and PRIMOS operating system subroutines which can be called from any point within a PL1G program. These routines are described in The PRIMOS Subroutines Reference Guide.

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 and more powerful programming tools and methods than any other language currently available.

THE PL/I SUBSET G LANGUAGE

Many programming situations exist for which the PL/I language is appropriate, but which do not require the more advanced, unusual, or rarely used features of full PL/I. To meet the needs of such situations, ANSI has extracted from full PL/I a subset language known as PL/I Subset G.

Subset G was designed by ANSI to have the following properties:

- It is well suited for use in commercial, scientific, and systems programming application areas.
- It is small enough to achieve widespread implementation yet large enough to achieve widespread usage.

- It is easier to learn and understand, substantially less expensive to implement, and utilizes computer resources more economically than full PL/I.

- Programs written in the subset are more portable than programs written in the full language.

- Programs written in the subset are less likely to contain programming errors than programs written in the full language because the subset is both simpler and less permissive than the full language.

Differences between Full PL/I and PL/I Subset G

A detailed comparison of full PL/I with PL/I Subset G is found in Appendix F.

THE PL1G LANGUAGE

PL1G is Prime's version of PL/I Subset G. It differs only minimally from standard PL/I Subset G. The differences consist of extensions to and exclusions from the standard language.

PL1G Extensions to PL/I Subset G

Prime has avoided extending PL/I Subset G unnecessarily, since needless extensions would serve mostly to reduce compatibility between PL1G and other versions of PL/I. Following is a complete list of all Prime extensions to PL/I Subset G.

- %REPLACE statements.

- RANK and BYTE built-in functions. (However, these functions can be easily expressed in terms of standard built-in functions.)

- READ and WRITE statements operating on stream files.

- Nonstandard properties of the device named TTY.

- Use of an A-format without a field width to read a variable length input line.

- Use of both upper- and lower-case letters in names. Some implementations maintain upper- and lower-case letters as distinct characters, while others require upper-case only.

- Use of the OPTIONS(SHORT) specifier to control space allocation for pointer variables.

- Use of %LIST; and %NOLIST; statements to control listing detail. These are equivalent to the IBM PL/I statements %PRINT; and %NOPRINT;.
- The -APPEND argument to the TITLE option.
- The characters # and @ may be used in identifiers.
- The DO statement accepts an index variable of any arithmetic type.
- Unconnected arrays may be passed to *-extent parameters.
- The ONLOC built-in function is provided.

None of these features should be used in any program which may have to run under some non-Prime version of PL/I.

PL1G Exclusions from PL/I Subset G

During the development of PL1G at Prime, standardization of PL/I Subset G continued at ANSI. The final revision of BSR3.74, Revision 8, included a few features which were originally not part of the subset. These features are not implemented in PL1G as of Revision 17.2.

The excluded features are:

- The assignment "array = scalar" is not allowed.
- The conditions FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, ZERODIVIDE, and UNDEFINEDFILE are not included.
- The STRING option is not allowed in a GET or PUT statement.
- READ with SET is not allowed.

Implementation-Defined Features of PL1G

The ANSI standard for PL/I Subset G does not specify every detail of the language. Certain features which 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 section of this guide. Specific details on Prime's choices for each such feature are contained in Section 11.

PL1G programs which may have to run under a non-Prime version of PL/I should be written to be minimally dependent on implementation-defined features.

Where appropriate, the description of an implementation-defined feature includes a discussion of factors relating to an implementation's choice for that feature. This information can be used in estimating the possible impact of an implementation-defined feature on program transportability. The information appears with each general description and is not reiterated in Section 11.

RESTRICTIONS ON PL1G PROGRAMS

The segmented nature of the Prime virtual memory system imposes a few restrictions on PL1G programs. None of them 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 PL1G compiler can reference or be referenced by modules produced by the FTN, F77, or COBOL compilers. Certain restrictions must be observed when a PL1G 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 PL1G should be declared as INTEGER in FORTRAN 77. See Appendix C for a description of PL1G 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 PL1G 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.

A PL1G static external structure may be used to reference a FORTRAN or PMA common block having the same name as the structure. Care must be taken that the data items in the structure and block correspond appropriately.

PL1G object modules can also interface with PMA (Prime Macro Assembler) routines. See The Assembly Language Programmer's Guide.

PL1G AND THE EDITOR

The characters ^ and ; have special meanings to the editor which conflict with their uses in PL1G. The ^ is the editor's escape character and PL1G's "not" character, while the ; is the editor's carriage return and PL1G's statement delimiter. A conflict arises whenever an attempt is made to enter either of these characters into a PL1G 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
```

where a and b must be single, currently non-special characters. The character a will replace the semicolon as a carriage return, and b will replace the up-arrow as an escape character. The semicolon and up-arrow are thereby freed for ordinary use.

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.

PL1G AND PRIME UTILITIES

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

- Multiple Index Data Access System (MIDAS)
- Forms Management System (FORMS)
- Database Management System (DBMS)

For complete information on any of these utilities, see the appropriate reference guide. Following is a brief description of MIDAS and FORMS. At Rev. 17.2, PL1G does not provide an interface with DBMS.

Multiple Index Data Access System (MIDAS)

MIDAS 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. Handling of indices, keys, pointers, and the rest of the file infra-structure is performed automatically for the user by MIDAS. Major advantages of MIDAS are:

- Large data files may be constructed
- Efficient search techniques
- Rapid data access
- Compatibility with existing Prime file structures
- Ease of building files
- Multiple user access to files
- Data entry lockout protection
- Partial/full file deletion utility

PL1G interfaces MIDAS through the use of MIDAS subroutine calls. Since MIDAS subroutines are written in Prime FORTRAN IV (FTN) the restrictions mentioned above under INTERFACE TO OTHER LANGUAGES apply.

See: Reference Guide, Multiple Index Data Access System (MIDAS).

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 which uses Prime's Input/Output Control System (IOCS), including programs written in PL1G. 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. To facilitate 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 PL1G with FORMS is identical to that of Prime FORTRAN IV (FTN), except that PL1G uses READ and WRITE on stream files - a Prime extension - to access FORMS.

See: FORMS Management System.

THE SOURCE LEVEL DEBUGGER

Prime makes available a powerful interactive debugging tool, the Source Level Debugger, which may be obtained by any Prime installation as a separately priced item. Use of the debugger can greatly expedite and simplify the debugging process. Major features of the debugger enable the programmer 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 subroutines or functions from debugger command level
- Trace statement execution
- Trace selected variables, printing a message when their value changes
- Print and/or change the value of any variable
- Print a subprogram call/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 Reference 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 returning 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. A programmer can specify his own response to a condition by supplying an on-unit of his own. When a condition occurs for which a programmer-supplied on-unit exists, the actions specified in the on-unit will be 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 PRIMOS Subroutines Guide.

CONVENTIONS USED IN THIS GUIDE

Various conventions are used in the following sections. Their meanings must be clearly understood by the reader.

Conventions Indicating Extensions

When a specific feature is explicitly described in the text as being a PL1G or Prime extension, the implication is that it is not part of PL/I Subset G. No such feature should be used in any program which may have to run under a non-Prime implementation of PL/I.

When the PL1G language is mentioned in general, the reference is to Prime's extended implementation of PL/I Subset G as a whole.

Every PL1G extension is listed above under "PL1G Extensions to PL/I Subset G."

Conventions in Examples

In all examples involving dialog between the user and the system, the user's input is underlined, and the system's output is not. For example:

```
OK, attach mydirec
OK, ed oldfile
EDIT
```

Examples consisting only of PL1G statements, with no responses from the system, are not underlined.

```
ZERO: PROCEDURE (ARG);
DECLARE ARG FIXED BIN;
ARG = 0;
RETURN;
END;
```

Typographical Conventions

WORDS-IN-UPPER-CASE	Uppercase letters identify command words or keywords. They are to be entered literally.
words-in-lower-case	Lowercase letters identify options or arguments. The user substitutes an appropriate numerical or text value.
Brackets [ ]	Brackets indicate that the item enclosed is optional.
Vertical slash |	Vertical slashes separate alternative options in an options list. Unless the list of options is enclosed by brackets, one option must be selected.
Parentheses ( )	When parentheses appear in a statement format, they must be included literally when the statement is used.
Ellipsis ...	An ellipsis indicates that the preceding item may be repeated.
Pound sign #	The pound sign is used in examples to indicate a blank.

Part II
The PL1G Language

SECTION 2

OVERVIEW OF PL1G

This section gives an overview of the PL1G language and defines such basic features as: program formatting rules, comment conventions, block structure and scope rules, as well as exception handling and input/output. The following sections are organized for easy reference and provide accurate and complete definitions of all language features.

PROGRAM TEXT

Programs are written by using a terminal to enter text and by using a text editor to correct and update the text. Programs produced on other computer systems or on card punch machines can be loaded into the computer from the card reader or magnetic tape.

Regardless of how the text is prepared and entered, it must not contain extraneous information such as sequence numbers or operating system control information.

The text may consist of variable length lines, as well as blank or empty lines and need not conform to any particular format. However, to improve the readability of the program text, the formatting rules suggested in Section 12 and used within the examples in this manual should be followed.

The program text that is input to the compiler is called a program module. The results of compiling several program modules may be bound together and executed as a single program.

NAMES, CONSTANTS, AND PUNCTUATION

The basic elements of the PL1G language are called tokens. A token is a name, a constant, a punctuation symbol, a comment, or a compile-time text modification statement as described later in this section.

Names

A name consists of a single letter, or a sequence of letters, digits, and the underscore character. The first character of a name must be a letter, and a name must not contain more than 32 characters. Examples:

```
EMPLOYEE_NAME

DELTA69

A
```

Names may be written using both lower and upper case letters. A lower case letter x is equivalent to an upper case X unless the -LCASE command line option has been given. See Section 14.

Names cannot contain blanks or hyphens. A blank serves to separate names that are not otherwise separated and a hyphen is the subtract operator. An underscore must be used in place of a hyphen.

Names are also known as identifiers and are referred to as such by error messages produced by the compiler. A name can be used to denote an object operated upon by the program such as a variable, file, or label. These names are known as declared names because their meaning is established by a declaration of the name. Names also are used to denote parts of statements such as verbs, options and clauses. A name used to denote part of a statement is called a keyword. Example:

```
DECLARE X(5) FIXED;

L:  X(3) = 25;
```

Both X and L are declared names, but DECLARE and FIXED are keywords.

There are no reserved names in PL1G. This means that names which are used as keywords such as READ or INTO in a read statement may also be used by the programmer as declared names. However, this practice makes programs difficult to read and should be avoided whenever possible.

Constants

A constant is a sequence of characters that represents a particular value. Examples:

```
25

7.5

3.12E-Ø1

'This is a character-string constant'

'He said, "I don''t know."'

'1'B

'1Ø11'B

'775'B3

'A7Ø'B4
```

The first example is an integer constant; the second is a fixed-point constant; the third is a floating-point constant; the next two are character-string constants; and the last four are bit-string

constants. The first two bit-string constants are written in binary notation; the next is written in octal notation, and the last is written in hexadecimal notation. Nonbinary representations of bit-string data are further discussed under BIT-STRING DATA in Section 3.

Arithmetic constants represent decimal values. Binary arithmetic values have no constant representation, but any decimal constant can be easily converted to a binary value by using it in a context that expects a binary arithmetic value.

A character-string constant can contain any character except '. If a ' is required within a character-string constant, it is written as ''. (The " character is not equivalent to ' or '' and has no more significance than any other character within a constant.)

Punctuation

Names and constants must be separated from one another by one or more blanks or by punctuation. Additional blanks may be used around punctuation but are not required.

Punctuation symbols are either operators or separators.

Operators are:

```
+ - * / **

= ^= < > <= >= ^< ^>

| (or !)  & ^

|| (or !!)  ->
```

Separators are:

```
() , . : ;
```

Examples:

```
A=B+C*D;

A = B+C *D;

DO K = N TO M BY K;
```

The first example requires no blanks because each name is separated from the next by a punctuation symbol. The second example shows the use of (or misuse of) extra blanks. The third example shows extra blanks around = but otherwise, it has the minimal number of blanks.

When an arithmetic constant is followed by a name, at least one blank is required to separate the arithmetic constant from the name.

In contexts where a character-string or bit-string constant follows a name, such as PICTURE '999', a blank is required by standard PL/I, but is not required by the PL1G compiler.

COMMENTS

A comment can appear anywhere that a blank can appear and is equivalent to a blank. The general form of a comment is:

```
/* Any sequence of characters except an asterisk followed
immediately by a slash */
```

Examples:

```
/*THIS IS A COMMENT*/

IF A>25 /* AND SO IS THIS */ THEN
```

If the */ is omitted from a comment, all program text up to the next */ is considered part of the comment. This causes part of the program to be ignored by the compiler and may cause it to produce misleading error messages. The same problem can occur if a character-string or bit-string constant is not terminated with a '.

The compiler puts an * (asterisk) into the line number field of the listing for each line on which a comment is continued from a preceding line. This asterisk can be used to determine if the program text was accidentally included in a comment.

TEXT REPLACEMENT AND INSERTION

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.

%INCLUDE

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 an operating system file name and its format is implementation defined.

%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.

%REPLACE

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 is normally used to supply the sizes of tables or to give names to special constants whose meaning would not otherwise be obvious. Examples:

```
%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. This means that 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. Block structure is discussed later in this section.

Also, %REPLACE is not part of standard PL/I and may not be available in other implementations of PL/I.

STATEMENTS

As words and punctuation symbols are used to form sentences in English, tokens are used to form statements in PL1G. A statement is a sequence of tokens ending with a semicolon. All statements, except the assignment statement, begin with a keyword that identifies the purpose of the statement. Examples:

```
READ FILE(F) INTO(X);

GOTO L;

CALL P(A,B,C);
```

```
RETURN;

STOP;

DO K = 1 TO 1Ø;

A = -C;  /* ASSIGNMENT STATEMENT */
```

Compound Statements

PL1G has two compound statements: the IF statement and the ON statement. They are called compound statements because they contain another statement. Examples:

```
IF A>B
   THEN READ FILE (F) INTO (X);

ON ENDFILE(F) STOP;
```

The general form of an IF statement is:

```
IF expression
   THEN statement
   [ELSE statement]
```

Since each statement ends with a semicolon, an IF statement needs no semicolon of its own. IF statements can be nested as shown by the following example:

```
IF A>B
   THEN IF D<C
        THEN READ FILE(F) INTO(X);
        ELSE STOP;
```

In this example, the second IF statement has both a THEN clause and an ELSE clause, because an ELSE always corresponds to the immediately preceding THEN. The first IF statement has only a THEN clause. If we want to stop when A^>B we must write a null ELSE clause. Example:

```
IF A>B
   THEN IF D<C
        THEN READ FILE(F) INTO (X);
        ELSE;
   ELSE STOP;
```

ON statements are compound statements but are not nested. They are used to respond to exceptional conditions and are discussed later in this section, and in Section 9. Example:

```
ON ENDFILE(F) STOP;
```

Order of Execution

Statements constitute a framework within which expressions are evaluated and values are assigned to variables. Statements also control the order of execution of the program.

Statements are normally executed in the order in which they are written. However, each statement can be labeled and a GOTO statement can be used to alter the normal order of execution. Example:

```
      A = 5;
      GOTO L2;
   L1:
      B = 7;
      .
      .
      .
   L2:
      B = 4;
```

In this example, A is assigned 5, and B is assigned 4. The statements beginning with label L1 can be executed only if a GOTO statement transfers control to the label L1.

Programs that contain numerous labels are more difficult to read and normally have more errors than programs that contain few labels. The use of labels and GOTOs can be avoided or minimized by using the IF and DO statements to control the order of execution of statements. Examples:

```
IF A>B
   THEN DO;
          .
          .
          .
        END;

DO WHILE(A>B);
   .
   .
   .
END;

DO K = 1 TO 10;
   .
   .
   .
END;
```

The DO statement causes all statements between the DO statement and its corresponding END statement to be executed a variable number of times depending on the form of the DO statement. See Section 9 for a complete discussion of DO statements.

DECLARATIONS AND REFERENCES

The meaning of a name is determined by a declaration of the name rather than by the contexts in which the name is used. There are two kinds of declarations in PL1G, the DECLARE statement and the label prefix.

Although it can appear anywhere within the program, except as part of a compound statement, a DECLARE statement is not an executable statement and has no effect except to establish clearly the meaning of names. Examples:

```
DECLARE F FILE;

DECLARE A(5) FLOAT;

DECLARE (I,J,K) FIXED BINARY(15);
```

F is declared as the name of a file; A is declared as a floating-point array variable; and I, J, and K are declared as integer variables.

A label prefix declares a name as a name of a format, or as the name of a procedure, or as a statement label. Examples:

```
P:  PROCEDURE;

F:  FORMAT(F(10,2),A(6));

L:  READ FILE(F) INTO(X);
```

P is declared as a procedure name, F as a format name, and L as a statement label. Label prefixes appearing on statements other than FORMAT and PROCEDURE statements are declarations of statement labels.

Use of a name in any context, other than in a declaration, constitutes a reference to the name. In order to determine the meaning of the reference, the compiler searches for a declaration of the name. This search resolves the reference by associating it with a declaration of the name. Example:

```
DECLARE A FIXED BINARY(15);
DECLARE B FLOAT;
    .
    .
    .
B = B+1;
    .
    .
    .
A = A+1;
```

The compiler knows that it must generate a floating-point add for B+1 because it has resolved the reference B to the second DECLARE statement. Likewise, it knows that it must generate an integer add for A+1 because it has resolved the reference A to the first DECLARE

statement. See Section 6 for a complete discussion of reference resolution.

PROCEDURES

A procedure is a sequence of statements beginning with a PROCEDURE statement and ending with an END statement. A procedure defines a block of statements and is sometimes called a block. Examples:

```
A:  PROCEDURE;
       .
       .
       .
    END A;


B:  PROCEDURE;
       .
       .
       .
    END B;
```

The preceding examples show two procedures A and B.

Procedures contained within other procedures are called nested or internal procedures. Procedures not contained within another procedure are called external procedures. A program module must consist of one or more external procedures. Example:

```
A:  PROCEDURE;
       .
       .
       .
    B:  PROCEDURE;
          .
          .
          .
        END B;

    C:  PROCEDURE;
          .
          .
          .
        D:  PROCEDURE;
              .
              .
              .
            END D;
        END C;
    END A;
```

Procedures B and C are nested within procedure A. Procedure D is nested within procedure C. Procedure A is external.

BLOCK STRUCTURE AND SCOPE

Each procedure establishes a distinct region of the program text called the scope throughout which names declared within that procedure are known. The scope of a name is therefore, that region of the program within which the name may be referenced.

The scope of a name includes the procedure in which it is declared and all procedures contained within that procedure, except those procedures in which the same name is redeclared. Example:

```
A:  PROCEDURE;
    DECLARE (X,Y) FLOAT;
       .
       .
       .
    B:  PROCEDURE;
        DECLARE X FILE;
        DECLARE Z FIXED;
           .
           .
           .
        END B;
    END A;
```

The scope of Y includes both procedure A and procedure B. The scope of X as a file is procedure B. The scope of Z is procedure B. Z cannot be referenced from within A.

PARAMETERS AND ARGUMENTS

The purpose of a procedure is to "package" a set of executable statements and declarations to form a block that can be executed from several places in the program simply by calling it.

The usefulness of a procedure is greatly increased if it can be made to operate on different values each time it is called. Example:

```
    CALL P(A,B);
       .
       .
       .
    CALL P(D,E);
       .
       .
       .
P:  PROCEDURE(X,Y);
       .
       .
       .
```

```
        PUT FILE (F) LIST(X,Y);
           .
           .
           .
        END P;
```

A and B are arguments of the first call to the procedure P. While P is executing as a result of the first call, the argument A is said to correspond to the parameter X and the argument B corresponds to the parameter Y. While P is executing as a result of the second call, D corresponds to X and E corresponds to Y.

CALLS AND RETURNS

The statements within a procedure are only executed when the procedure is called from another procedure. Example:

```
    A:  PROCEDURE;
           .
           .
           .
        CALL B;
           .
           .
           .
        END A;
           .
           .
           .
    B:  PROCEDURE;
           .
           .
           .
        END B;
```

Procedure B is executed when it is called from procedure A. Procedure A resumes execution when B is completed.

A procedure returns to its caller either by executing its END statement or by executing a RETURN statement. Example:

```
    A:  PROCEDURE;
           .
           .
           .
        IF X<Ø THEN RETURN;
           .
           .
           .
        END A;
```

BLOCK ACTIVATION AND RECURSION

Each time a procedure is called, it is said to be active and it remains active until it returns from the call. Each such procedure activation has an associated block of storage allocated on a stack. This block of storage is called a stack frame and it is used to hold information that is unique to each procedure activation, such as the location to which control should return from the procedure activation.

If procedure A calls procedure B, the stack frame associated with the activation of A is pushed down by the stack frame associated with the activation of B. When B returns, its stack frame is removed from the stack and the stack frame of A is then the current stack frame.

If procedure A calls itself or calls another procedure that calls A, A is said to be a recursive procedure.

Recursive procedures must be written with the RECURSIVE option. Example:

```
A:  PROCEDURE RECURSIVE;
        .
        .
        .
    CALL A;
        .
        .
        .
    END A;
```

Recursive procedures are a powerful tool for use in applications that process data bases or other list-structures. Because recursive procedures do not exist in most other popular languages, they are unfamiliar to many programmers.

The following example is a program that makes a copy of a list structure. The example uses BASED variables and other concepts that are discussed in later sections of this manual. Example:

```
      .
      .
      .
NEW = COPY(OLD);
      .
      .
      .
COPY: PROCEDURE(IN) RETURNS(POINTER) RECURSIVE;

      DECLARE (IN,OUT)   POINTER;
      DECLARE 1 RECORD   BASED,
              2 FIELD1   FLOAT,
              2 FIELD2   FLOAT,
              2 SON      POINTER,
              2 DAUGHTER POINTER;
```

```
DECLARE NULL BUILTIN;

IF IN = NULL THEN RETURN(NULL);
ALLOCATE RECORD SET(OUT);
OUT->RECORD.FIELD1 = IN->RECORD.FIELD1;
OUT->RECORD.FIELD2 = IN->RECORD.FIELD2;
OUT->RECORD.SON = COPY(IN->RECORD.SON);
OUT->RECORD.DAUGHTER = COPY(IN->RECORD.DAUGHTER);
RETURN(OUT);

END COPY;
```

Each record of the original list structure may be linked via pointers to a son record and/or a daughter record. Either the son or the daughter field may contain a null pointer value. COPY is called with a pointer to the top record in the original list structure. It returns a pointer to the top record in a new list structure which is a copy of the original. Each activation of the procedure COPY has its own instance of the variables IN and OUT.

Additional information about recursive procedures is given in Section 3 which discusses entry data.

VARIABLES AND STORAGE

A variable is a named object that is capable of holding values. Each variable has two properties that determine what kind of value it holds and how long it holds them. A variable's data type determines what kind of values the variable holds, and a variable's storage class determines how long it holds them.

Data Types and Conversion

Each variable must be declared to have a data type. Examples:

```
DECLARE A FLOAT DECIMAL;

DECLARE N FIXED BINARY(15);

DECLARE C CHARACTER(30);

DECLARE B BIT(1);
```

A is capable of holding any floating-point decimal value, N is capable of holding any binary integer value, C is capable of holding any string of 30 characters, and B is capable of holding either '1'B or '0'B.

Failure to specify a data type causes the compiler to issue an error message and to give the variable a data type of FIXED BINARY.

Although each variable is only capable of holding values of a specific

data type, the language provides a complete set of operations that convert values from one data type to another. Whenever a value is assigned to a variable, it is first converted to the data type of the variable. Examples:

```
A = 1;      /* converts 1 to FLOAT */

N = 2.5;    /* converts 2.5 to integer 2 */

C = -4.5;   /* converts -4.5 to a character string */

B = Ø;      /* converts integer Ø to 'Ø'B */
```

A complete discussion of data types is given in Section 3, and a complete discussion of conversions is given in Section 8.

Storage Classes

A variable's storage class determines when and for how long storage is to be allocated for the variable. Since this storage holds the variable's values, the storage class of a variable determines how long the variable retains its values.

Unless declared otherwise, a variable is given the AUTOMATIC storage class. This means that the variable is allocated storage each time that its containing procedure is called. Every time the procedure returns to its caller, the storage allocated for that call is freed. Consequently, AUTOMATIC variables do not retain their values after their containing procedure returns.

If a variable must retain its value between calls of its containing procedure, it should be declared to have the STATIC storage class. Examples:

```
DECLARE K FIXED BINARY STATIC;

DECLARE TABLE(4) CHARACTER(5)
        STATIC INITIAL('A','B','C','D');
```

Each element of the array TABLE is given an initial value. Only STATIC variables can be given initial values.

Storage for STATIC variables is allocated prior to program execution by the compiler and/or loader. Consequently, the size of each STATIC variable must be specified as an integer constant.

The additional storage classes BASED and DEFINED are explained in Section 4. That section also provides more detailed information on STATIC and AUTOMATIC storage.

BEGIN BLOCKS

A BEGIN block is exactly like a procedure except that: it is executed by executing its BEGIN statement, it cannot have any parameters, it always returns by executing its END statement, and execution of a RETURN statement within a BEGIN block returns to the caller of the procedure that immediately contains the BEGIN block. Example:

```
BEGIN;
   .
   .
   .
END;
```

Because of its limitations, a BEGIN block is less useful than a procedure and is normally not necessary. However, because a BEGIN block serves to define the scope of any declarations that it contains, it can be used to contain declarations and executable statements that reference those declarations. In this manner a BEGIN block can be used to redeclare a name that has been declared in a containing block and can be used to cause allocation and freeing of the AUTOMATIC variables declared within the block. Example:

```
DECLARE INDEX FIXED BINARY;
   .
   .
   .
BEGIN;
DECLARE INDEX FILE;
DECLARE X(1000) FLOAT;
   .
   .
   .
END;
```

INDEX is redeclared within the BEGIN block so that it can be used as a file. A large array X is declared within the BEGIN block and is allocated storage only during the execution of the BEGIN block.

EXCEPTION HANDLING

The ON statement gives the program the ability to respond to exceptional conditions such as end-of-file, or computational errors such as division by zero. Examples:

```
1.  ON ENDFILE(F)
    BEGIN;
      .
      .
      .
    END;

2.  ON ENDFILE(G) STOP;
```

```
3.  ON ERROR
    BEGIN;
      .
      .
      .
    END;
```

The general form of an ON statement is:

```
ON condition-name on_unit
```

condition-name is ERROR, ENDFILE(f), ENDPAGE(f), or KEY(f). on-unit is any statement except: IF, ON, PROCEDURE, RETURN, DECLARE, DO, or END.

Execution of an ON statement establishes the on-unit as a block of statements that are executed if the specified condition occurs. It does not cause immediate execution of the on-unit.

An on-unit is established within its containing block's current activation and remains established until that block returns to its caller or until another on-unit is established for the same condition within the same block activation.

If one of the possible conditions occurs during the execution of a block and that block does not have an established on-unit for that condition, the calling block's on-unit is used to respond to the condition. If the calling block has no established on-unit for the condition, its caller's on-unit is used. If no ancestor has an on-unit for the condition, a default action is taken. Except for the ENDPAGE condition, this default action terminates program execution and issues an execution-time error message.

An on-unit for the ERROR condition can determine which of many possible errors caused the condition by using the ONCODE built-in function explained in Section 10. On-units for the ERROR condition cannot resume execution of the statement in which the error was detected. Normally, an on-unit should be written to print debugging information and execute a STOP statement, or execute a GOTO statement to continue execution of the program elsewhere.

Additional information on exception handling is given in Section 9 under the ON Statement.

INPUT AND OUTPUT

PL1G provides two types of input and output: stream I/O and record I/O. Each type of I/O has its own statements and each type operates on its own kind of files.

Files

A stream file is a sequence of characters organized into lines and pages. A record file is a set of discrete records that are either accessible sequentially or accessible by character-string valued keys. Each record in a keyed file must have a unique key-value.

Each record of a record file may be of a different length from other records in the file. The maximum length of a record and the length of keys is implementation defined and given in Section 11.

A file is referenced by using a file name declared with the FILE attribute. Example:

```
DECLARE F FILE;
```

F is a file name or a file constant. It is called a constant because it cannot be the target of an assignment operation. Associated with each file constant is a block of static storage called a file control block in which information about the current status of the file is retained while the file is open. The form and content of this information is of no interest to the PL1G programmer, but the programmer must be aware that the file constant F is in effect a pointer to or a designator of a file control block.

A file variable is a variable that can be assigned file values. That is, it can be assigned a file constant or the value of another file variable that previously had been assigned a file constant. Example:

```
DECLARE G FILE;
DECLARE F FILE;
DECLARE V FILE VARIABLE;

V=F;
V=G;
```

F and G are file constants each of which has an associated file control block. V is a file variable that can be assigned file values. After the first assignment, both F and V designate the same file control block. Any operation performed on V is equivalent to the same operation performed on F. After the second assignment, any operation performed on V is equivalent to the same operation performed on G because both V and G designate the same file control block.

A file name used as a parameter is a file variable and designates the same file control block as its corresponding argument.

Each file control block has a file-id that is the name of its associated file constant. This file-id is used as the value of the ONFILE built-in function and as the default TITLE option used when opening the file.

Complete information for each I/O statement is given in Section 9.

File Opening and Closing

A file is normally opened by executing an OPEN statement that gives the name of the file as understood by the operating system as well as the properties or attributes that the file is to have during the time that it is open.

Refer to Section 9 for a discussion of OPEN statements.

Examples:

```
1. OPEN FILE(F) TITLE('ALPHA_NEW')
          RECORD OUTPUT;

2. OPEN FILE(G) KEYED INPUT;

3. OPEN FILE(CONSOLE) TITLE('TTY -DEVICE')
          STREAM INPUT;

4. OPEN FILE(PTR) TITLE('PR1 -DEVICE')
          STREAM OUTPUT PRINT LINESIZE(132)
          PAGESIZE(6Ø);
```

The FILE option specifies a file value. This may be either a file constant, file variable, or file-valued function as explained under FILE DATA in Section 3.

The file value specified by the FILE option is subsequently used by I/O statements to operate on the file.

The TITLE option gives the name by which the operating system knows the file. The TITLE option may specify devices and may also contain any additional information needed by the operating system. Refer to the OPEN Statement in Section 9 for the specific format of the TITLE option.

If no TITLE option is given, a default TITLE is made from the file-id.

The properties that the file is to have while it is open are chosen from the following sets:

```
STREAM [INPUT|OUTPUT [PRINT]]

RECORD [DIRECT|SEQUENTIAL] [INPUT|OUTPUT|UPDATE] [KEYED]
```

Certain attributes imply others. An attribute does not have to be specified if it is implied by an attribute that is specified.

Attribute	Implied Attributes
PRINT	OUTPUT STREAM
DIRECT	RECORD KEYED
KEYED	RECORD
SEQUENTIAL	RECORD
UPDATE	RECORD

Certain attributes are required. If not explicitly supplied or not implied, they are supplied by default.

Required Attribute	Default
STREAM or RECORD	STREAM
INPUT or OUTPUT or UPDATE	INPUT
SEQUENTIAL or DIRECT	SEQUENTIAL, if RECORD

A LINESIZE option can be specified for any STREAM OUTPUT file. If none is specified, a default line size is assumed. The default may vary according to the file name or device. A PAGESIZE option can be specified for a PRINT file. If none is specified, a default page size is supplied.

If a file is operated upon by an I/O statement prior to being opened, it is opened implicitly and given a set of default attributes. It has a default TITLE taken from the file-id and it has attributes determined by the kind of statement used to open the file.

Statement	Opening Attributes
GET	STREAM INPUT
PUT	STREAM OUTPUT
READ	RECORD INPUT
WRITE	RECORD OUTPUT
REWRITE	RECORD UPDATE
DELETE	RECORD UPDATE

Implied attributes and default attributes are also supplied in an attempt to make a complete set of attributes. Because defaults are not obvious and can produce the wrong attributes, programmers should always use an OPEN statement to explicitly open a file and should supply sufficient attributes to ensure correct opening.

File attributes can also be declared with a file constant in a DECLARE statement. Any such attributes are merged with the attributes supplied by the opening to form the set of file attributes for a given opening. The programmer must ensure that file attributes declared in a DECLARE statement are valid for all openings of the file control block that is associated with the file constant for which the attributes were declared.

A file is closed by executing a CLOSE statement or when the program executes a STOP statement. Once closed, a file control block can be used to open another file, or the same file may be opened again with possibly different attributes. Example:

```
OPEN FILE(F) TITLE('ALPHA_FILE')
     RECORD OUTPUT;

WRITE FILE(F) FROM(X);
   .
   .
   .
CLOSE FILE(F);
   .
   .
   .
OPEN FILE(F) TITLE('ALPHA_FILE')
     RECORD INPUT;
   .
   .
   .
READ FILE(F) INTO(X);
```

Stream I/O

Stream files are written by PUT statements and are read by GET statements. Transmission of data to or from a stream file under control of a format-list is called edit-directed I/O. Transmission of data to or from a stream file without a format-list is called list-directed I/O. Format-lists can be part of the GET or PUT statement or can be given by a FORMAT statement.

Execution of a GET or PUT statement transmits data to or from the current line of a stream file but does not generally produce or consume an entire line. Several PUT statements can be used to build a given line or a single PUT may build one or more lines. Likewise, several GET statements may read values from a given line or a single GET statement may read one or more lines.

Each stream file has an associated line size that is determined when the file is opened. Lines of an output file contain n characters where n is the line size of the file. However, a shorter line can be created by use of a SKIP option on a PUT statement. Example:

```
PUT FILE(F) SKIP LIST(A,B,C);

PUT FILE(F) SKIP;
```

This example causes the current line to be output. Then one or more new lines containing the values of A, B, and C is created. The second PUT statement forces the last line containing A, B, and C to be output and starts a new line. The SKIP option is always executed before any data is transmitted.

On input, each line may be of any length up to the value of the line size. Execution of a GET statement reads from the current line until it is empty, then reads a new line and continues reading values and lines until the list of variables has been read. A SKIP option can be used to force a new line to be read prior to reading any values. Example:

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

GET FILE(F) LIST(D);

GET FILE(F) SKIP LIST(X,Y);
```

In this example, A, B, and C are read from the current line or from as many lines as are necessary. D is read from the same line as C, unless C happened to be the last value on its line. The SKIP option reads a new line and consequently ignores any values remaining on the line after the value received by D. X and Y are read from the line read by the SKIP as well as any additional lines that are necessary.

A PAGE option can be used in a PUT statement to begin a new page prior to transmitting any values. Example:

```
PUT FILE(F) PAGE LIST(A,B,C);
```

A complete discussion of GET and PUT statements as well as FORMAT statements is given in Section 9.

An output stream file opened with the PRINT attribute has a current line number whose value can be accessed using the LINENO built-in function. Each time a line is written to the output stream, the line number is incremented by one. This occurs regardless of what caused the lines to be written. Each new line is initially positioned so that the next item is written to column 1 of the line.

An output stream file opened with the PRINT attribute has a current page number and a page size. The current page number can be read using the PAGENO built-in function and can be set using the PAGENO pseudo-variable. The ENDPAGE condition is signalled when the line to be written has a line number that is page size plus one.

Each time a new page is written, the line number is reset to one and the page number is incremented by one.

If an on-unit for the ENDPAGE condition does not write a new page, the line number is allowed to increase indefinitely until a new page is written. The ENDPAGE condition is only signalled when the line to be written has a line number one greater than the page size.

A new page is only written by a PAGE option of a GET statement, by a PAGE format-item, and by the default on-unit for the ENDPAGE condition.

Control characters such as carriage return, new line, form feed,

vertical tab, horizontal tab, ring bell, null, etc. must not be written as data characters to a stream file and cannot be read as data. In some implementations, these characters may be used to mark line and page boundaries, but in other implementations, boundaries may be unmarked.

List-directed I/O: Values transmitted by list-directed I/O are separated by blanks on output and may be separated by blanks or commas on input. Each value is transmitted in a readable format determined by the value's data type. This is the most convenient kind of I/O for many applications. Example:

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

If file F has been opened as STREAM OUTPUT (but not as PRINT), A is an integer declared FIXED BINARY(15) and has a current value of -75, and B is a character-string declared CHARACTER(5) whose current value is DOG##, this example produces:

```
######-75#'DOG##'#
```

in the current output line of file F. The field ######-75 is produced by conversion of the FIXED BINARY(15) integer to a character-string (the larger number of blanks allows for the maximum possible value plus sign and decimal point and leading Ø for fractional values as explained in Section 8).

A single blank separates the value of A from the value of B, that is, 'DOG##'. The last blank separates B from the next field.

If F had been opened as PRINT, enough blanks would have been supplied at the end of each item to align the next item at an implementation defined tab stop. Also, the single quotes around the value of B would have been omitted.

For input, list-directed I/O considers each blank or comma to terminate a field. Excess blanks within a field to be assigned to an arithmetic variable are ignored. Such fields must simply contain a valid constant as could be written in the text of a program.

A field to be assigned to a character-string variable may contain a constant as written in the text of a program or it may contain a sequence of any characters. In the latter case, the sequence begins with the first nonblank character in the field and ends with the character immediately preceding the next comma or blank. Example:

```
GET FILE(F) LIST(A,B);
```

If A is FIXED BINARY(15) and B is CHARACTER(5), the following lines have the indicated effect on the values of A and B:

Line	A	B
5,'abc'#	5	abc##
-45##'abc'#	-45	abc##
-45#,#abc-4#	-45	abc-4

Edit-directed I/O: Values transmitted by edit-directed I/O are transmitted into fixed length fields whose length and content are controlled by data-formats. Each data-format corresponds to an element in the value list of the GET or PUT statement and causes that value to be converted and transmitted. Example:

```
PUT FILE(F) EDIT(A,B) (F(7,2),A(6));
```

If A is declared FIXED BINARY(15) and has a current value of -45, and B is declared CHARACTER(4) and has a current value of DOG#, the following is written into the current line of file F:

```
#-45.ØØDOG###
```

Control formats can be used to force new lines, skip parts of lines, etc. Example:

```
PUT FILE(F) EDIT(A,B) (SKIP,F(7,2),X(3),A(6));
```

Using the same values of A and B as our previous example, this statement produces:

```
#-45.ØØ###DOG###
```

in a new output line.

Edit-directed input requires that each input line be exactly described by the controlling format. If the current input line contains fewer characters than are required to satisfy the format, additional lines are read until the format is satisfied. Example:

```
GET FILE(F) EDIT(X) (A(8Ø));
```

If X is declared as CHARACTER(1ØØ) and the current line contains only 6Ø characters: the 6Ø characters are read, a new line is read, and 2Ø additional characters are read from that line. The 8Ø characters thus read are then assigned to X and 2Ø blanks are used to pad the value of X.

Record I/O

Record files are read and written using READ, WRITE, REWRITE, and DELETE statements. Execution of one of these statements transmits one record to or from the file and updates the current position of the file.

Record files with keys can be opened as SEQUENTIAL, KEYED SEQUENTIAL, or DIRECT files. The method of opening determines the operations that are permitted on the file while it is open.

Opened As	Operations Allowed
SEQUENTIAL	READ|WRITE without key
KEYED SEQUENTIAL	READ|WRITE|REWRITE|DELETE with optional KEY
DIRECT	READ|WRITE|REWRITE with required KEY

A record file is opened for INPUT, OUTPUT, or UPDATE. The method of opening determines the operations that are permitted on the file while it is open.

Opened As	Operations Allowed
INPUT	READ
OUTPUT	WRITE
UPDATE	READ|WRITE|REWRITE|DELETE

If a file opened for INPUT does not exist, an error is signalled.

If a file opened for OUTPUT or UPDATE does not exist, a file is created. If the file has been opened as DIRECT or KEYED SEQUENTIAL, a keyed file is created; otherwise, a nonkeyed file is created.

If a file opened for OUTPUT already exists, it is deleted and a new file is created, unless -APPEND appears in the TITLE option.

A SEQUENTIAL file with or without keys always has a current position that is advanced to the next record by a READ or WRITE, but which is not advanced by a REWRITE or a DELETE. After opening for INPUT or UPDATE, files are positioned ahead of the first record and unless repositioned by a KEY option, the first operation on the file must be a READ.

Record I/O statements copy the storage of a variable to or from a record in a file. No conversion is performed and no check is made to ensure that the data being read is of the proper type to store into the variable. The variables used in INTO or FROM options cannot be unaligned bit-strings or structures consisting entirely of unaligned bit-strings, because such variables normally share a portion of their storage with other members of the same array or structure. Also note that an expression cannot be used in a FROM option. Examples:

```
READ FILE(F) INTO(X);

READ FILE(F) KEY(N+N) INTO(Y);

WRITE FILE(G) FROM(X);

WRITE FILE(G) KEYFROM(N+M) FROM(Y);
```

```
REWRITE FILE(H) KEY(N+M) FROM(X);

DELETE FILE(H) KEY(N+M);
```

A REWRITE statement replaces an existing record in an UPDATE file with a new record. A DELETE statement removes an existing record from an UPDATE file.

A complete discussion of record I/O statements is found in Section 9.

SECTION 3

DATA AND DATA TYPES

DATA TYPES

Each value has a data type that determines which operations can be performed on the value and how the value is represented in storage. Data types are declared for variables and function results using the following attributes:

Arithmetic Data	String Data	Other Data
FIXED BINARY	PICTURE	POINTER
FIXED DECIMAL	CHARACTER	LABEL
FLOAT BINARY	CHARACTER VARYING	ENTRY
FLOAT DECIMAL	BIT	FILE
	BIT ALIGNED	

These attributes are also used to declare named constants such as files and external procedures that are part of another program module, and they are used within the ENTRY attribute to describe the data type of each parameter of a procedure. See Section 5 for a discussion of the ENTRY attribute.

The data type of a value produced by an expression is determined by the operators and the built-in functions within the expression. Section 7 gives the data type produced by each operator and Section 1Ø gives the data type of the values produced by each built-in function.

The data type of a constant is determined by the syntax of the constant.

A variable or value having one of these data types is called a scalar variable or scalar value, unless it is an array. In that case, it is called an array variable or array value.

ARITHMETIC DATA

Each arithmetic value has a base that is either binary or decimal, a scale that is either fixed-point or floating-point, and a precision that is the number of digits in the value. These three properties collectively constitute the data type of arithmetic values and can be expressed using the following attributes:

```
FIXED BINARY(p)
FIXED DECIMAL(p,q)
FLOAT BINARY(p)
FLOAT DECIMAL(p)
```

Each implementation of PL/I Subset G imposes limits on the maximum

values of p and q. These limits normally are different for each combination of base and scale.

Fixed-point Data

Fixed-point numbers contain p digits. For fixed-point decimal numbers, q of those digits may be fractional digits. Fixed-point binary numbers are always integers. Examples:

```
DECLARE K FIXED BINARY(15);

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

The values of K are fixed-point integers with 15 binary digits. Therefore, K can hold any value in the range -32767 to +32767 or -(2**15)+1 to (2**15)-1.

The values of D are fixed-point numbers with 7 decimal digits, two of which are fractional digits. Therefore, D can hold any value in the range -99999.99 to +99999.99

Because most computers calculate addresses of data using binary integers, programmers should use FIXED BINARY variables whenever possible. The relative efficiency of FIXED BINARY variables used as subscripts, string lengths, do index variables, etc. is very significant.

FIXED DECIMAL values are normally used only when fractional digits are required. The representation of FIXED DECIMAL values within storage depends on the implementation, but it is normally packed decimal data that cannot be directly used in address calculations and that normally requires more computer time to process than does binary data.

Except for division, all operations on fixed-point data align the decimal points and produce results that retain all fractional digits, and as many integral digits as can be supported by the implementation. The divide operator's result has a precision that provides for all possible integral quotient digits plus as many fractional digits as can be supported by the implementation. In most implementations, an attempt to calculate a fixed-point value larger than can be supported results in a signal of the ERROR condition.

In the full PL/I language, operations involving both decimal and binary values always produce a binary result value. However, because binary fractions are not permitted in subset PL/I, any operation that produces a fractional result produces a warning diagnostic from the compiler and a decimal result.

An implementation may retain more than p digits when representing fixed-point values in storage. However, a program that produces values of more than p digits is invalid and may or may not produce consistent results when run on another implementation of PL/I. If a fixed-point value is converted to a character-string or bit-string, the length of

the string is determined by p, not by the actual value. See Section 8 for a discussion of the conversion rules.

Fixed-point values are never rounded unless explicitly rounded by the use of a built-in function. Assignment to a fixed-point variable causes truncation of excess low order digits. All possible roundings can be performed by use of the CEIL, FLOOR, TRUNC, or ROUND built-in functions.

Fixed-point decimal values are always stored so as to accurately represent decimal fractions. A fixed-point decimal value of 10.50 is never represented as 10.49.

Fixed-point constants are written as decimal numbers with or without a decimal point. Examples:

```
5

4.5

4100.01
```

If a fixed-point constant contains a decimal point, it is considered to be a scaled fixed-point value and is stored and accessed like a FIXED DECIMAL variable. Fixed-point constants without a decimal point are integer constants and can safely be used in operations with any other arithmetic value regardless of the other value's base or scale. Programmers are advised to always write integer constants without a decimal point.

The precision of a fixed-point constant is the number of digits in the constant.

Floating-point Data

Floating-point numbers consist of a mantissa m, a base b, and an exponent e. The number is given by:

```
m*b**e
```

The mantissa m is a fraction containing at least p digits. The value of b and the possible range of e are defined by each implementation. However, if the base is binary, m contains the equivalent of at least p binary digits, and if the base is decimal, m contains the equivalent of at least p decimal digits. Examples:

```
DECLARE X FLOAT BINARY(23);

DECLARE Y FLOAT DECIMAL(7);
```

In this example, the values of X are floating-point numbers whose mantissa contains the equivalent of at least 23 binary digits. The values of Y are floating-point numbers whose mantissa contains the

equivalent of at least 7 decimal digits.

The representation of floating-point values in storage depends on the implementation. An implementation may represent the mantissa in any base that it chooses providing that it contains an equivalent of at least p binary or p decimal digits. On some computers, decimal floating-point values are represented by using a decimal mantissa, but most implementations use a binary or hexadecimal mantissa for all floating-point numbers.

Unless the implementation represents decimal floating-point numbers differently than it represents binary floating-point numbers, the choice of binary or decimal is not important. In that case, the programmer should specify decimal precisions that suit the problem being solved. (Refer to Appendix C for a discussion of data formats.)

Because floating-point values may be represented in any base (usually not decimal) and because excess digits may be lost during calculations, floating-point values are approximate. However, any integer value that is converted to floating-point and converted back to integer retains its original value. Likewise, the floating-point calculations of add, subtract, and multiply of integer values produce integer values.

Floating-point constants are written as a fixed-point constant followed by an exponent. Examples:

```
5E+Ø2

4.5E1

1ØØE-Ø4

.ØØ1E-Ø4

ØEØ
```

Floating-point constants have a decimal precision of p, where p is the number of digits in the fixed-point constant. For example, the second constant given above has a precision of 2.

When fixed-point constants such as 1.5 are used in operations with floating-point values, they should be written with an exponent in order to avoid conversion of fixed-point decimal values to floating-point.

If floating-point values are converted to character-strings or bit-strings, the length of the resulting string is determined by p, not by the actual value. See Section 8 for a discussion of conversion rules.

PICTURED DATA

A value whose data type is determined by a PICTURE attribute or by the P format in a FORMAT statement is called a pictured value. Pictured

values are character-string values that represent fixed-point decimal numbers that may contain embedded symbols such as . , $ CR DB etc. Examples:

```
DECLARE A PICTURE '$ZZ,ZZZV.99CR';

DECLARE B PICTURE '$$,$$$V.99-';

DECLARE C PICTURE '$**,***V.999';

DECLARE D PICTURE '----9V.999';

DECLARE E PICTURE '9999V9999S';
```

If the value 1234.56 were assigned to each of these variables, the resulting pictured values would be:

Variable	Pictured Variable
A	$#1,234.56##
B	#$1,234.56#
C	$*1,234.560
D	#1234.560
E	12345600+

Note

The pound sign (#) in the preceding (and subsequent) examples of pictured data represents a space.

The purpose of each picture character is explained in the following list:

Character	Interpretation
V	Indicates the position of the decimal point. All digit positions to the right of the V are fractional digits. Any value assigned to a pictured value is first scaled so that its decimal point is aligned with the V.
B , . /	These picture characters are inserted into the pictured value only if they are preceded by a non-zero digit or if they are preceded by a 9 or V picture character. If they are not inserted, a zero suppression character of blank or * is inserted instead. The picture character B denotes a blank rather than the the letter B.
Z	Causes zero suppression using a blank as the the suppression character. A Z must not be preceded by a 9 or a drifting field. A picture containing a Z must not also contain an *.

*	Causes zero suppression using * as the the suppression character. An * must not be preceded by a 9 or a drifting field. A picture containing an * must not also contain a Z.
S	If an S occurs more than once in a picture, the entire field of S's is a drifting field and can only contain a V and one or more B , . or / picture characters. Such a field cannot be preceded by a 9, Z, or * and if it contains a V followed by one or more S's, it cannot be followed by a 9. The total number of digits represented by a drifting field is one less than the number of S's in the field. The digits are zero suppressed and a sign of + or - is inserted immediately preceding the most significant digit. A single S causes a sign of + or - to be inserted into the pictured value.
+	Operates exactly like S, except that the sign of negative values is indicated by a blank.
-	Operates exactly like S, except that the sign of positive values is indicated by a blank.
$	Operates like S, except that a $ is inserted instead of the sign.
9	Causes zero suppression to stop and a digit to be inserted into the pictured value.
CR	These two characters must appear as a pair and may only appear on the rightmost end of the picture. They are replaced by two blanks if the value is not negative.
DB	Exactly like CR

A picture cannot contain more than one sign character + - S CR or DB unless all such sign characters are part of a drifting field.

When a zero value is assigned to a pictured variable and the picture does not contain at least one 9, the entire pictured value is filled with blanks or with * depending on whether or not the picture contained any * picture characters.

When a nonzero value is assigned to a pictured variable and the V is followed by zero suppression characters or by part of a drifting field, the V stops zero suppression. Examples:

```
DECLARE A PICTURE 'ZZZV.ZZ';

DECLARE B PICTURE '***V.**CR';
```

A value of .Ø1 assigned to A and B produces ###.Ø1 and ***.Ø1##. A value of zero assigned to A and B produces ###### and ********.

If a V is not given, one is assumed to be on the rightmost end of the picture.

Each 9, Z, or * is considered to be a digit of precision. Within a drifting field each S, +, -, or $, except the first one, is considered to be a digit of precision. All digits of precision following the V are fractional digits. Examples:

```
DECLARE A PICTURE 'ZZZV.ZZ'

DECLARE B PICTURE '---V.--'
```

A has a precision of (5,2) while B has a precision of (4,2).

Negative values cannot be assigned to a pictured variable unless the picture contains a CR, DB, -, or S.

Values assigned to pictured variables are truncated and aligned with the V as they would be when assigned to a FIXED DECIMAL variable of the equivalent precision.

Used in contexts that expect an arithmetic value or used with a relational operator, pictured values are converted to fixed-point decimal values with a precision p and q that are determined by the picture.

A pictured value is operated upon as if it were a character-string value only when it is assigned to a character-string variable, and when it is an operand of the concatenate operator or of a built-in function that expects character-string operands. In all other contexts, pictured values are operated upon as if they were fixed-point decimal values.

CHARACTER-STRING DATA

A character-string value is a sequence of characters. The number of characters in a sequence is called the length of the value.

A character-string of zero length is called a null string.

A character-string variable or character-string valued function is declared with these attributes:

```
CHARACTER(n)
    or
CHARACTER(n) VARYING
```

n is an integer-valued expression that specifies the maximum length of all string values that can be held by the variable or returned by the function.

The VARYING attribute causes the string variable or function to hold or return values of varying lengths. The representation of a varying string variable in storage is such that any string up to n characters may be held by the variable, and the length of the current string is retained as part of the value.

Without the VARYING attribute, a string variable or function always holds or returns values of length n. An assignment to a nonvarying string always extends short values with blanks on the right to make them n characters long.

Assignments of a string of more than n characters to either a VARYING or a nonvarying string variable causes only the leftmost n characters to be assigned and excess characters to be truncated.

Character-string values are compared from left-to-right using the collating sequence of the computer. Strings of unequal length are compared by extending the shorter string with blanks on the right.

Nonvarying character-string variables always occupy exactly n bytes of storage. As elements of arrays or members of a structure, they begin on the next available byte and are not aligned on word or other storage address boundaries. This permits an array of nonvarying characters to be stored and accessed as if it was a single string. See Section 4 which discusses storage sharing.

A character-string constant is written as:

```
'Any characters except quote'
```

If a quote (apostrophe) is required within the constant, it must be written as a pair of quotes. Examples:

```
'ABC'

'He said, "I don''t know."'

''
```

The second example shows a pair of quotes used within the word don't. It also shows the insignificance of the double quote as a delimiter of a character-string constant. The last example is a null string.

BIT-STRING DATA

A bit-string is a sequence of bits. The number of bits in the sequence is called the length of the value.

A bit-string of zero length is called a null string.

A bit-string variable or bit-string valued function is declared with these attributes:

```
BIT(n)
    or
BIT(n) ALIGNED
```

n is an integer valued expression which specifies the maximum length of all string values that can be held by the variable or returned by the function.

If a value of less than n bits is assigned to a bit-string variable, the variable is extended on the right with zero bits to make it n bits long. If a value of more than n bits is assigned, only the leftmost n bits are used and the excess bits are truncated.

The ALIGNED attribute causes the bit-string variable for which it is declared to be aligned in storage so as to make it more efficiently accessed. It may also cause the variable to occupy more than n bits, but does not increase the size of values that can be stored in the variable. The ALIGNED attribute has no effect on operations performed on the variable, but is considered part of the data type for purposes of argument/parameter matching and storage sharing as described in Section 4.

Bit-string values are compared from left to right a bit at a time until an inequality is found. The shorter operand is extended with zero bits on the right to make it the length of the other operand.

A bit-string is not a word of storage in a computer and is not an arithmetic value. It is a sequence of bits that is always operated upon from left to right just as a character string is operated upon from left to right.

However, an unaligned bit-string variable always occupies exactly n bits of storage. As elements of arrays or members of structures, they always begin on the next available bit and are not aligned on byte, word, or other storage address boundaries. This permits a structure containing unaligned bit-strings to be used to describe objects such as system control tables, machine instructions, etc. commonly used by systems programmers.

A bit-string of length 1 is a boolean value with the possible values of Ø meaning false and 1 meaning true.

A bit-string constant is written as: 'Zeros and Ones'B. Examples:

```
'1'B

'1Ø11Ø'B

'Ø'B

''B
```

The last example is a null bit-string.

Bit-string constants may also be written using a string of characters to represent the bit-string:

```
'character string'Bn
```

Where n is 1, 2, 3, or 4 and is the number of bits each character represents. Table 3-1 gives the permitted set of characters and shows the corresponding bit-string values. Examples:

```
'Ø73'B3

'AØ4'B4
```

The first example is equivalent to 'ØØ111Ø11'B and the second example is equivalent to '1Ø1ØØØØØØ1ØØ'B. (The first, 'Ø73'B3, represents an octal value, and the second, 'AØ4'B4, represents a hexadecimal value.)

POINTER DATA

A pointer value is the address of a variable's storage. Example:

```
DECLARE A(5) FIXED BINARY(15);
DECLARE P POINTER;
   .
   .
   .
P = ADDR(A(K));
```

P is a pointer variable that is capable of holding the address of any PL1G variable. The assignment statement uses the ADDR built-in function to calculate the address of A(K) and assigns that address as the value of P.

A pointer is used with a template to access the storage to which it points. Example:

```
DECLARE X FIXED BINARY(15) BASED;
   .
   .
   .
P->X = 1Ø;
```

Table 3-1.
Permitted Set of Characters and
Corresponding Bit-String Values

	B or B1	B2	B3	B4
Ø	Ø	ØØ	ØØØ	ØØØØ
1	1	Ø1	ØØ1	ØØØ1
2	illegal	1Ø	Ø1Ø	ØØ1Ø
3	"	11	Ø11	ØØ11
4	"	illegal	1ØØ	Ø1ØØ
5	"	"	1Ø1	Ø1Ø1
6	"	"	11Ø	Ø11Ø
7	"	"	111	Ø111
8	"	"	illegal	1ØØØ
9	"	"	"	1ØØ1
A	"	"	"	1Ø1Ø
B	"	"	"	1Ø11
C	"	"	"	11ØØ
D	"	"	"	11Ø1
E	"	"	"	111Ø
F	"	"	"	1111

X is a BASED variable or template describing a fixed-point binary integer. P is the pointer variable from our previous example. The assignment assigns 1Ø to A(K).

Pointers are normally used to locate the storage of dynamically allocated BASED variables as described in Section 4, rather than used as a mechanism for accessing another variable's storage as is shown by these two examples.

The null pointer value is produced by the NULL built-in function. It is a unique value that does not address any variable and is used to indicate that a pointer variable does not currently address anything. Example:

```
DECLARE CHAIN_HEAD POINTER;
   .
   .
   .
CHAIN_HEAD = NULL();
```

Pointer values may be assigned, compared for equality or inequality, or passed as arguments, and returned from functions, but no calculations or conversions can be performed on them. They cannot be transmitted in stream I/O.

If the variable whose storage is addressed by a pointer value is freed, the pointer value can no longer be used. Use of such a pointer to access the storage causes unpredictable results.

Because most computers cannot directly address unaligned bit data, the compiler produces a warning diagnostic when the argument of the ADDR built-in function is an unaligned bit string. However, this warning can be ignored if portability is not a concern.

The BASED variable that is used as a template normally must have the same data type as the variable addressed by the pointer. Violations of this rule cause unpredictable results. Programs that violate this rule and produce "correct" results, may fail when compiled with optimization enabled or may fail when moved to another implementation of PL/I. See Section 4 for a discussion of storage sharing by BASED variables.

There is no constant representation of a pointer value. However, the NULL built-in function can be used in any context, including an INITIAL attribute, where a constant might be desired.

LABEL DATA

A label prefix on a statement other than a PROCEDURE or FORMAT statement is a declaration of a statement label.

The LABEL attribute can be used to declare variables and functions whose values are labels. Example:

```
A:  PROCEDURE;
    DECLARE L LABEL;
       .
       .
       .
    L1:
       .
       .
       .
    L = L1;
```

The label variable L is assigned a statement label value. A subsequent GOTO L; would transfer control to the statement labeled L1.

A label value consists of two parts. One part designates a statement (the instructions compiled for the statement) and the other part designates a stack frame of the block that immediately contains the statement. In the previous example, the assignment of L1 to L assigns a designator to the statement labeled L1 as part one of L, and assigns a designator to the current stack frame of procedure A as part two of

L. The execution of a subsequent GOTO uses the stack frame designator only if the GOTO is executed in a block activation other than the one whose stack frame is designated by the label value. The value of a label variable is invalid and must not be used if its stack frame designator refers to a block that is no longer active. Example:

```
A:  PROCEDURE;
    DECLARE L LABEL;
      .
      .
      .
    L1:
      .
      .
      .
    L = L1;
        .
        .
        .
    GOTO L;
        .
        .
        .
    B:  PROCEDURE;
          .
          .
          .
        GOTO L;
            .
            .
            .
```

Execution of the first GOTO simply transfers control to L1 because it occurs within the same block activation as assigned L1 to L (and consequently the stack frame component of L designates the stack frame that is current when the GOTO is executed). However, execution of the second GOTO occurs within a block activation other than the one that assigned to L. Execution of the second GOTO first uses the stack frame designator of L to terminate the block activation of B and restore the block activation of A; then it transfers control to the statement labeled L1.

Label prefixes may be subscripted by a single optionally signed integer constant. Example:

```
            GOTO CASE(K);
    CASE(1):
                .
                .
                .
    CASE(2):
                .
                .
                .
    CASE(3):
                .
                .
                .
    CASE(6):
                .
                .
                .
```

CASE is a one-dimensional array of statement labels containing 6 elements. Elements 4 and 5 are undefined and cannot be used. Use of an undefined element produces unpredictable results.

Label prefixes on PROCEDURE and FORMAT statements cannot be subscripted.

Because the label array is declared by label prefixes, it cannot also be declared as a LABEL variable by appearing in a DECLARE statement.

Label values may be assigned, compared for equality or inequality, passed as arguments, and returned from functions, but no calculations or conversions can be performed on them. They cannot be transmitted in stream I/O.

An array of statement labels, such as CASE in our previous example, cannot be used as an array value, but its elements can be used in any context that permits a statement label.

ENTRY DATA

The label prefix on a PROCEDURE statement is a declaration of a procedure name. Names of external procedures that are not part of the compiled module must be declared using an ENTRY attribute and, if they are functions, a RETURNS attribute.

The keyword ENTRY is used rather than PROCEDURE because in full PL/I, procedures can be entered at points other than the PROCEDURE statement. Examples:

```
    DECLARE E ENTRY(FIXED BINARY(15), POINTER);
    DECLARE F ENTRY((5) FLOAT DECIMAL(7));
              RETURNS(FLOAT DECIMAL(7));
```

E is declared as a procedure name that requires two arguments and which

must be called as a subroutine by a CALL statement. F is declared as a procedure name which must be referenced as a function. F requires an array of five FLOAT DECIMAL(7) values as its argument and returns a FLOAT DECIMAL(7) result.

Procedures, other than external procedures that are not part of the compiled module, are declared by their PROCEDURE statement and cannot be declared in a DECLARE statement.

Failure to declare an external procedure that is part of another program module results in error messages from the compiler.

The ENTRY attribute together with the VARIABLE attribute can be used to declare variables whose values are entry values. The ENTRY attribute can be used in a RETURN attribute to indicate that a function returns entry values. Example:

```
A:  PROCEDURE;
       .
       .
       .
    DECLARE E ENTRY VARIABLE;
       .
       .
       .
    E = A;
```

E is an entry variable that is capable of being assigned any entry value. The assignment assigns the procedure name A as the value of E. A subsequent call to E would call A.

Any attributes specified within an ENTRY attribute and any RETURNS attribute specified for an entry variable have no effect on the values that can be assigned to the variable. However, when the entry variable is called, the value that it currently holds must designate a PROCEDURE statement whose parameters and RETURNS option match those given in the declaration of the entry variable.

In our previous example, a program that calls E as a function or calls E with arguments is invalid and produces unpredictable results.

Like a label value, an entry value consists of two parts. The first part designates a PROCEDURE statement and the second part designates a stack frame of the block that immediately contains the PROCEDURE statement. An ENTRY value that designates an external procedure has a null second part. An entry value is invalid and must not be used if its stack frame designator refers to a block that is no longer active.

Entry values may be assigned, compared for equality or inequality, passed as arguments, and returned from functions but no calculations or conversions can be performed on them. They cannot be transmitted in stream I/O.

Unless a programmer uses entry variables or entry parameters and also

uses recursive procedures, he can ignore the entire discussion of the second part of an entry value and can consider an entry value to be simply a designator of an entry point. The discussion that follows explains the way in which the second part of an entry value is determined and how it is used.

If a procedure references variables that are declared in its containing procedure and those variables are allocated in that procedure's stack frame, the inner procedure must be given a designator to the outer procedure's stack frame in order for it to be able to access the outer procedure's variables. The second part of an entry value is this stack frame designator.

Except when the outer procedure has been activated recursively, only one stack frame for the outer procedure exists and that is the one that is always used when the inner procedure references variables declared in the outer procedure.

If the outer procedure has been activated recursively, the entry value used to call the inner procedure determines which stack frame is to be used when the inner procedure references the outer procedure's variables. Example:

```
A:  PROCEDURE RECURSIVE;
    DECLARE X FIXED BINARY(15);
    DECLARE E ENTRY VARIABLE STATIC;
    IF FIRST_TIME
       THEN E = B;
       ELSE CALL F;
    CALL G;
      .
      .
      .
    B:  PROCEDURE;
          .
          .
          .
        X = 5;
```

Assume that A calls G, and that G calls A, then A calls F, and F calls the entry value held by E. When E is called, it results in an activation of B because B was assigned to E. When that activation of B references X, two stack frames containing an instance of X exist. Two stack frames exist because two calls of A are still active. However, because the first of those two activations of A assigned B to E, E contains a designator to the first stack frame of A. Consequently, B assigns to the first instance of X. If B were called directly from A, B would always use the most recent stack frame of A when referencing X.

An older stack frame is only used when an entry name is assigned to an entry variable or passed as an argument to an entry parameter. When the entry name is thus passed or assigned, the current stack frame of its containing block is assigned as the second component of the entry value produced by that assignment.

If the procedure name being assigned or passed as an argument is not declared within the block that is assigning or passing it, but is declared in a containing block, the containing block's stack frame is found using the same principle as was used to find X in our example. Once found, a designator to that stack frame is assigned as part two of the entry value.

Each stack frame for a block A has a designator to a stack frame of the block that contains A. That designator is the designator which was supplied as the second part of the entry value used to activate A. Unless the activation of A was the result of calling an entry variable or entry parameter and unless recursion has been used, the stack frame thus designated is the most recent instance of the block immediately containing A.

FILE DATA

A file value designates a file control block that can be opened or closed and which can thereby be connected to various files and devices known to the operating system. Example:

```
DECLARE F FILE;
```

F is a file constant that designates a file control block. That file control block may be opened, closed and used to perform I/O on files and devices known to the operating system. See Section 2 for a discussion of PL1G I/O.

A file variable is a variable that is capable of being assigned any file value, and it is declared using both the FILE and VARIABLE attributes. Example:

```
DECLARE G FILE VARIABLE;
DECLARE F FILE;
   .
   .
   .
   G = F;
```

G is a file variable and is assigned the value of the file constant F. After the assignment, operations on G are equivalent to operations on F because they both designate the file control block belonging to F.

File values may be assigned, compared for equality or inequality, passed as arguments, and returned from functions, but no calculations or conversions can be performed on them. They cannot be transmitted by stream I/O.

ARRAYS

An array is an ordered set of values all having the same data type. Elements of an array are referenced by their position within the array. For this purpose, each array has a specified number of dimensions and each dimension has a specified lower and upper bound. Examples:

```
DECLARE A(1:4) FIXED BINARY(15);

DECLARE B(0:3,0:5) FLOAT BINARY(23);

DECLARE C(-2:10) CHARACTER(5);

DECLARE D(25,4,2) POINTER;
```

A is a one-dimensional array capable of holding fixed-point binary values. The lower bound is 1, the upper bound is 4, and the number of elements is four.

B is a two-dimensional array capable of holding floating-point binary values. Both lower bounds are zero, and the upper bounds are 3 and 5. Therefore, B has four rows of six columns each.

C is a one-dimensional array capable of holding character-string values each of which has five characters. Its lower bound is -2 and its upper bound is 10 giving it a total of 13 elements.

D is a three-dimensional array capable of holding pointer values. Because no lower bounds are given, they are assumed to be 1. The array has a total of 200 elements organized as 25 sets of four rows, each of which has two columns.

As is indicated by these examples, array elements are stored in row-major order. That means they are stored such that when accessed in the order in which they are stored, the rightmost subscript varies most rapidly and the leftmost subscript varies least rapidly.

The maximum number of dimensions is eight, but the maximum values of the bounds depend on the implementation.

The bounds of AUTOMATIC, DEFINED or BASED arrays may be specified by integer-valued expressions as explained in Section 4. The bounds of STATIC arrays must be integer constants. The bounds of parameter arrays can either be integer constants or *. If an * is given as the bound of a parameter array, it represents both the lower and upper bound and means that the actual bounds of the corresponding array argument are to be used as the bounds of the array parameter. Example:

```
DECLARE A(0:5) FIXED BINARY;
CALL P(A);
   .
   .
   .
P:  PROCEDURE(X);
    DECLARE X(*) FIXED BINARY;
      .
      .
      .
```

During the call of P, the bounds of X are (0:5) and any reference to X is a reference to A.

Elements of an array are referenced using as many subscripts as the array has dimensions. Example:

```
DECLARE A(-2:5,4,3) FIXED BINARY;
```

A reference to A(-2,1,1) is a reference to the first column of the first row in the first set. A reference to A(3,2,1) is a reference to the first column of the second row in the sixth set.

A subscript must be an integer valued expression. The use of fixed-point fractions or floating-point values as subscripts produces an error message from the compiler. However, such values can be converted to integer values by use of the TRUNC, CEIL, FLOOR, or ROUND built-in functions.

Each subscript must lie within the range specified by its corresponding lower and upper bound. Unless subscript range checking has been requested, the compiler does not produce code to check the range of subscript values. If checking is requested, any subscript that exceeds its range results in a signal of the ERROR condition.

Entire arrays may be transmitted in stream or record I/O, passed as arguments, and assigned to other arrays of the same size and shape, but no conversions or calculations can be performed on entire arrays. Subscripted references to array elements can be used in any context that permits a variable reference.

STRUCTURES

A structure is a hierarchically ordered set of values that may be of different data types. The immediate components of a structure are called members of the structure. A structure that is itself a member of another structure is called a substructure. A structure that is not a substructure is called a major structure.

The hierarchical organization of a structure is specified by using level-numbers as shown by the following example:

```
DECLARE 1 S,
          2 A FIXED BINARY,
          2 B FLOAT BINARY,
          2 T,
              3 P POINTER,
              3 Q CHARACTER(10);
```

S is a major structure or level-one structure. All major structures must have a level-number of one. The members of S are A, B, and T. T is a substructure whose members are P and Q.

Members normally have a level-number that is one greater than their containing structure. However, they may be given any level number that is greater than the level-number of their containing structure but which is not greater than the level-numbers of their fellow members. Example:

```
DECLARE 1  S,
          20 A FIXED BINARY,
          20 B FLOAT BINARY,
          20 T,
              30 P POINTER,
              30 Q CHARACTER(10);
```

This structure is equivalent to the previous example.

An entire structure may be transmitted in stream or record I/O, passed as an argument, or assigned to another structure of identical size and shape and having members of corresponding identical data types, but no conversions or calculations can be performed on entire structures.

Members of structures can be referenced in any context that permits a reference to a variable. If a member's name is not otherwise declared in the same scope (that is: the block containing the reference, the block containing the declaration of the member, or any intervening block), the member may be referenced by its name alone.

The name of a member must be unique within its immediately containing structure, but may be used as the name of members of other structures or as the name of a nonmember. If a member's name has been used for more than one object, each reference to the member must be qualified by the name of its containing structure. If the containing structure's name has been used for more than one object, the name of its containing structure must be used as a qualifier. The names of major structures cannot be reused, only member names can be reused. Example:

```
DECLARE A FIXED BINARY;
DECLARE 1 S,
          2 A FLOAT BINARY,
          2 T,
              3 A POINTER;
```

A reference to A is a reference to the fixed-point variable. A reference to S.A is a reference to the floating-point variable. A

reference to T.A or S.T.A is a reference to the pointer variable.

A good programming practice is always to use the major structure name as a qualifier and avoid using the same member name more than once anywhere within the major structure or any contained structures. See Section 6 for more information on the resolution of references.

ARRAYS OF STRUCTURES

Structures, like other variables, can be declared as arrays and may contain arrays as members. Example:

```
DECLARE 1 S(5),
          2 A    FIXED BINARY,
          2 B(4) FLOAT BINARY,
          2 T(3),
            3 P    POINTER,
            3 Q(6) CHARACTER(10);
```

S is an array of structures that contains five structures. Each structure contains a fixed-point member followed by an array of four floating point values, followed by an array of substructures containing three substructures. Each substructure contains a pointer variable followed by an array of six character-string variables.

The entire structure contains five occurrences of A, 20 occurrences of B, 15 occurrences of T, 15 occurrences of P, and 90 occurrences of Q.

Each member of an array of structures is an array because there exist as many instances of the member as there are elements in the array. If a member is an array in its own right because it has its own dimension information (as do B, T, and Q in our example), the array inherits the additional dimensions from its containing structures. In our example, Q is a three-dimensional array whose bounds are (5,3,6).

Any member of a dimensioned structure is an array and must be referenced using a subscript for each of its dimensions. The subscripts may be written anywhere within the structure qualified reference, but it is a good practice to write each subscript immediately following the name to which it applies.

Using our previous example, S(K).A is a reference to the Kth element of A; S(K).B(J) is equivalent to B(K,J); S(K).T(J).Q(I) is equivalent to S.T.Q(K,J,I) or to S.T(K,J).Q(I), or S(K,J,I).T.Q, etc.

Members of dimensioned structures can be used as arrays only in stream I/O and as arguments to array parameters whose bounds have been specified as asterisks. They cannot be assigned, used in record I/O, used in ADDR or DEFINED, or passed to parameters with constant bounds.

Subscripted references to members of dimensioned structures can be used in any context that permits a variable reference.

SECTION 4

STORAGE CLASSES

STORAGE CLASSES

Every variable has a storage class that determines how and when storage is allocated for the variable. Except for parameters, each variable can be declared to have one of these storage classes:

AUTOMATIC
STATIC
BASED
DEFINED

If no storage class is specified for a variable that is not a parameter, AUTOMATIC is supplied by default. A parameter is recognized as having the parameter storage class by its appearance in a PROCEDURE statement.

The declared length of a string variable and the bounds of an array variable are called extents. The extents of a variable determine the variable's storage size and are evaluated when storage is allocated for the variable. The permitted forms for extents differ for each storage class and are described in the discussion of each storage class.

AUTOMATIC STORAGE

Storage is allocated for an automatic variable each time that the procedure or the BEGIN block in which it is declared is activated. Storage is allocated within the stack frame that represents that block activation. If a block is activated recursively, each stack frame contains an instance of all automatic variables declared in that block.

When a block activation terminates, its stack frame is removed from the stack thus freeing the storage of all automatic variables allocated within it. This normally occurs when a procedure returns to its caller or when a BEGIN block executes its END statement. However, it also occurs when a GOTO statement transfers control back to a previous block activation. In that case, all block activations between the block to which control is returned and the current block are terminated and their stack frames are popped from the stack.

Extents of automatic variables may be specified as integer-valued expressions and must not contain references to other automatic and defined variables declared in the same block.

Extent expressions of automatic variables are evaluated each time the containing block is activated. Their values are saved in the stack frame and effectively fix the size of the automatic variable for that block activation. Subsequent assignment to a variable used as an

extent does not affect the size. Example:

```
DECLARE A(N) FIXED;
   .
   .
   .
N = 1Ø;
```

The size of A is determined upon entry to the containing block by evaluating N and storing its value in the stack frame. The assignment to N does not change the size of A. Upon entry to the block, N must have a correct value and N must not be an automatic or defined variable declared in this block. Use of the HBOUND built-in function within this block activation would return the original value of N that was saved in the stack frame.

STATIC STORAGE

Storage is allocated for a static variable by the compiler and/or loader prior to program execution. Storage remains allocated throughout program execution.

Only one instance of a static variable's storage is allocated regardless of whether or not it is declared in a recursive procedure. Values assigned to a static variable are retained between calls to the variable's procedure.

The INITIAL attribute explained in Section 5 can be specified in the declaration of a static variable.

All extents specified for a static variable must be constants.

Static variables can be declared with the INTERNAL or EXTERNAL attribute. If neither is specified, INTERNAL is assumed.

An internal static variable is known in its containing block and all contained blocks, except blocks in which the same name is redeclared. This is the normal scope rule that applies to all other variables.

An external static variable is known in its containing block and all contained blocks, except blocks in which the same name is redeclared. However, all declarations of the same name that have the EXTERNAL attribute share the same storage and must specify identical attributes including any initial values. External static variables are known and shared by all program modules in a manner similar to the common variables of FORTRAN, except that each external static variable is shared independently of others.

BASED STORAGE

A based variable differs from all other variables in the sense that it is a template or description of storage, but does not have a block of storage of its own. Because it has no storage, it has no address.

In order to reference storage using a based variable, we must supply the address of the storage we want to reference. The based variable serves as a description of the storage but a pointer value must be used to supply the address of that storage. Example:

```
DECLARE X CHARACTER(3Ø) BASED;
DECLARE P POINTER;
  .
  .
  .
P->X = 'ABC';
```

Assume that we have assigned an address or pointer value to P. The reference P->X is called a pointer-qualified reference and it enables us to assign to the storage addressed by P as if that storage contained a character-string variable like X.

There are two ways in which we can obtain a pointer value: the ADDR built-in function can calculate the address of a variable's storage and return it, or an ALLOCATE statement can dynamically allocate a block of storage and give us a pointer to it. Example:

```
DECLARE X(5) FLOAT BINARY;
DECLARE Y FLOAT BINARY BASED;
DECLARE P POINTER;
DECLARE K FIXED BINARY(15)
  .
  .
  .
P = ADDR(X(K));
P->Y = 1Ø;
```

The ADDR built-in function calculates the address of X(K) and assigns it to the pointer P. The second assignment assigns 1Ø to the storage addressed by P and effectively assigns 1Ø to X(K).

The use of based variables to access another variable's storage is generally not a good programming practice because when reading the program it may be difficult to remember just what storage a given pointer addresses. There is also a danger that the based variable does not accurately describe the storage addressed by the pointer. Our previous example would be incorrect if Y were declared with a data type that differed from that declared for X. See the discussion later in this section for a treatment of storage sharing by based variables.

The ALLOCATE statement uses a based variable to allocate a block of storage that can later be accessed by a based variable together with the pointer returned by the ALLOCATE statement. Example:

```
DECLARE X(1Ø) FLOAT BASED;
DECLARE (P,Q) POINTER;
  .
  .
  .
ALLOCATE X SET(P);
  .
  .
  .
ALLOCATE X SET(Q);
```

The first ALLOCATE statement allocates a block of storage of sufficient size to hold an array of 1Ø floating-point values and assigns a pointer to that block of storage to P. P->X thereafter is a reference to that block of storage as an array of 1Ø floating-point values. The second ALLOCATE statement allocates a similar block of storage and assigns a pointer to the block as the value of Q. Q->X can then be used to reference the second block and P->X can be used to reference the first block.

Based variables can be subscripted just like any other variable. If we want to refer to the elements of our two arrays, we can write references like P->X(K) or Q->X(5). Since the pointer variable may also be an array it can be subscripted. Example:

```
DECLARE P(3) POINTER;
DECLARE X(1Ø) FLOAT BASED;
  .
  .
  .
ALLOCATE X SET(P(K));
```

P(K)->X is a valid reference to the entire array of floating-point values, and P(K)->X(J) is a valid reference to the Jth floating-point value. P->X is invalid because the pointer must not be an array value.

The ALLOCATE statement is the only context in which a based variable can be used without an explicit or implicit pointer qualifier. In this context, the based variable is used to describe how much storage is to be allocated.

Implicit pointer qualification is a short-hand notation that can be used in cases where all or nearly all references to a based variable use the same pointer. Example:

```
DECLARE X(1Ø) FLOAT BASED(P);
DECLARE P POINTER;
  .
  .
  .
ALLOCATE X SET(P);
```

Because X was declared to be based on P, unqualified references to X

such as X(K), X(5) or X are implicitly qualified by P and are equivalent to P->X(K), P->X(5), and P->X. Any explicit qualification such as Q->X may be used and is unaffected by the specification of P in BASED(P).

The extents of a based variable may be specified as integer-valued expressions. The extents are evaluated for each reference to the based variable and are not captured when storage is allocated by an ALLOCATE statement. It is the programmer's responsibility to ensure that any extents accurately describe the storage being referenced. Example:

```
DECLARE X(N) CHARACTER(1) BASED;

N = 10;
ALLOCATE X SET(P);
  .
  .
  .
N = M;
P->X(5) = 'A';
```

The ALLOCATE statement allocates an array of 10 elements. The value of M must be greater than or equal to 5 and less than or equal to 10 or the reference to P->X(5) is invalid.

A block of storage previously allocated by an ALLOCATE statement can be freed by a FREE statement. Example:

```
FREE P->X;
```

Once freed, the storage can no longer be accessed and any attempt to use a pointer such as P that points to the block produces unpredictable results.

When used with the ALLOCATE statement, based variables are a powerful tool for use in applications where the number of instances of a table or record are not known upon entry to a block and must be determined as the program executes. List structures consisting of based structures linked together via pointers provide a very high level of dynamic data structure for use in these situations. See the example of block activation and recursion in Section 2.

DEFINED STORAGE

A variable declared with the DEFINED attribute is an alternative description of the variable specified in the DEFINED attribute. Example:

```
DECLARE X CHARACTER(5) DEFINED(A);
DECLARE A(5) CHARACTER(1);
```

X is a character-string variable whose values are all of length 5. It shares storage with and is an alternative description of the array A.

For example:

```
X = 'abcde';
```

assigns 'a' to A(1), 'b' to A(2), and so on.

The extents of a defined variable may be integer-valued expressions. They are evaluated upon block entry and stored in the stack frame just like the extents of automatic variables. Such an extent expression must not contain a reference to any automatic or defined variable declared in the same block.

For additional information on defined variables, see the discussion in this section of the methods for giving alternative descriptions of storage.

PARAMETERS

A parameter has no storage of its own, but shares storage with its argument. Example:

```
DECLARE X FLOAT;
CALL P(X);
  .
  .
  .
P:  PROCEDURE (Y);
    DECLARE Y FLOAT;
```

During the call to P, X and Y describe the same storage. Where this occurs, the argument is said to have been passed by-reference.

If the argument is an expression, function reference, built-in function reference, constant, parenthesized variable reference, or a reference to a variable whose data type does not match that of the parameter, the argument is copied to a temporary block of storage in the caller's stack frame and is said to be passed by-value. In this case, the parameter is a description of that temporary storage.

The extents of a parameter can either be integer constants or asterisks. If they are integer constants, any argument that is to be passed by-reference must have identical constant extents. If the parameter's extents are asterisks, its corresponding argument may have extents of any value. In this case, the extents of the parameter are those of its corresponding argument. Example:

```
DECLARE A(10) FLOAT;
DECLARE B(25) FLOAT;
    .
    .
    .
```

```
CALL P(A);
    .
    .
    .
CALL P(B);
    .
    .
    .
P:  PROCEDURE(X);
    DECLARE X(*) FLOAT;
```

During the first call, the extents of X are (1:1Ø). During the second call, they are (1:25).

A character or bit string parameter with a constant length may be passed string arguments of variable length or differing constant length, but all such arguments are passed by-value and effectively converted to the length of the parameter.

An array parameter with constant extents can only be passed array arguments that have identical constant extents and which have an identical data type. A parameter structure can only be passed a structure or substructure of identical size, shape, and component data types. This means that array or structure arguments cannot be passed by-value.

The precision of arithmetic data is part of its data type. This means that a FLOAT DECIMAL(7) variable cannot be passed by-reference to anything except a FLOAT DECIMAL(7) parameter. VARYING and ALIGNED as well as the declared string length are part of a string variable's data type and must match those of the corresponding parameter. A picture variable can be passed by-reference only if the corresponding parameter has an identical picture.

If a reference to a variable appears as an argument enclosed within parentheses, it is considered to be an expression and is passed by-value. That is, it is assigned to a temporary block of storage that has the data type of the corresponding parameter.

If a reference to a variable appears as an argument and does not match the data type of the corresponding parameter, it is passed by-value unless it is a reference to an entire array or structure. The latter cases are invalid and produce an error message from the compiler. The former case produces a warning message that can be suppressed by enclosing the variable reference in parentheses. Example:

```
CALL P(A,(B));
```

A is passed by reference if it matches its corresponding parameter and is passed by-value if it does not. B is always passed by-value.

STORAGE SHARING

PL1G provides two storage classes, BASED and DEFINED, that are explicitly designed to permit a block of storage to be shared by several variables.

In all cases of storage sharing by BASED or DEFINED variables, storage can be shared only if:

- The data types of all variables sharing storage are identical, or
- All variables sharing storage are nonvarying character strings, or
- All variables sharing storage are unaligned bit-strings.

The latter two cases are called string overlays and permit the sharing variables to be arrays of any dimensionality and extents, or structures containing only the required type of string data, or scalar string variables. These dissimilar variables may share storage using the string overlay case because storage is known not to contain any gaps or extraneous information - it contains only characters in the second case and contains only bits in the third case. Example:

```
DECLARE X(4,4) CHARACTER(1);
DECLARE Y(16) CHARACTER(1) DEFINED(X);
DECLARE Z CHARACTER(5) DEFINED(X);
DECLARE 1 S BASED,
            2 A CHARACTER(8),
            2 B CHARACTER(8);
```

X has a block of storage containing 16 characters that is entirely shared by Y. Z shares the first five of these characters. S could be used to access all 16, while S.A could be used to access the first eight, and S.B could be used to access the last eight.

A based character string of more than 16 characters could not be used to access the storage, but one of 16 or fewer could be used.

A similar example could be constructed using all unaligned BIT data.

Storage sharing of other data types requires that the data types exactly match, but an element of an array may be shared with a nonarray variable of the same data type. Likewise, a member of a structure may be shared with a nonmember of the same data type.

Arrays may share storage with other arrays only if they have the same data type, same number of dimensions, and same extents. Structures may share storage with other structures only if they are left-to-right equivalent to the structure whose storage is being shared.

Left-to-right equivalent means that the sharing structures must be valid descriptions of the left part of the storage being shared. In

order to be a valid description of the left part of storage, they must have identical members up to and including all members contained anywhere within the last level-two item being shared. If any part of a level-two item is to be shared, all of it must be shared. Example:

```
DECLARE 1 S,          DECLARE 1 T BASED,        DECLARE 1 U BASED,
          2 A,                  2 A,                      2 A,
          2 B,                  2 B,                      2 B,
              3 C,                  3 C,                      3 C;
              3 D,                  3 D;
          2 E;
```

A reference to T.B.C is a valid reference to S.B.C, but a reference to U.B.C is not because the declaration of U does not describe all of the level-two item S.B.

A picture variable can only share storage with other pictured variables that have identical pictures.

As is the case for argument/parameter matching, the ALIGNED and VARYING attributes as well as the declared length of a string variable are part of its data type and must match, unless the strings qualify for string overlay sharing.

The base, scale, and precision of arithmetic variables must always match if they are to share storage.

SECTION 5

DECLARATIONS AND ATTRIBUTES

DECLARATIONS

Each name, except the name of a built-in function, must be declared either by a DECLARE statement or by a label prefix.

Each declaration has a scope or region of the program in which a reference to the name is associated with the declaration. The scope of a declaration includes the block in which it is declared and all contained blocks except blocks in which the name has been redeclared.

A name declared with the EXTERNAL attribute has the same scope rule as any other name, except that the object identified by that name is unique throughout the entire program. All declarations of a given name that have the EXTERNAL attribute identify the same object. Only files, static variables, and names of external procedures can have the EXTERNAL attribute. Files and procedures acquire the attribute by default, but static variables have internal scope unless they are explicitly declared with the EXTERNAL attribute.

A given name cannot be declared more than once within the same block, unless it is declared as the name of a structure member. In that case, the name may be redeclared within the same block providing that no two immediate members of the same structure have the same name and providing that the name is not declared more than once as a nonmember within the same block.

See Section 2 for a discussion of block structure and scope.

LABEL PREFIXES

A label prefix declares a name as a procedure name, format name, or statement label depending on the type of statement to which the prefix is attached.

A label prefix on a PROCEDURE statement declares a name as a procedure name. The name is declared in the block which contains the PROCEDURE statement, not the block defined by the PROCEDURE statement. The declaration contains a description of each parameter referenced by the PROCEDURE statement, as well as the data type specified by any RETURNS option of the PROCEDURE statement. Example:

```
E:  PROCEDURE(A,B) RETURNS(POINTER);
    DECLARE A FIXED BINARY;
    DECLARE B FLOAT DECIMAL;
```

E is declared as a procedure name in the block which contains the

PROCEDURE statement. The attributes of that declaration are RETURNS(POINTER) and ENTRY(FIXED BINARY, FLOAT DECIMAL).

The label prefix of a PROCEDURE or FORMAT statement cannot be subscripted.

The label prefix on a FORMAT statement declares a name as a format name. The name is declared in the block that contains the FORMAT statement. A format name is not a statement label and cannot be used in a GOTO statement. It can only be used in a format. See Section 9 for a discussion of FORMAT statements.

A label prefix attached to any other statement declares a name as a statement label. The declaration is established in the block that contains the statement to which the prefix is attached.

A label prefix attached to a BEGIN statement is declared in the block which contains the BEGIN statement, not the block defined by the BEGIN statement.

Label prefixes attached to statements other than PROCEDURE or FORMAT may be subscripted by a single optionally signed integer constant. Within a given block, all occurrences of a given name used in this manner must be subscripted, and all such prefixes collectively constitute a declaration of the name as an array of statement labels. Example:

```
                GOTO CASE(K);
CASE(1):
                .
                .
                .
CASE(2):
                .
                .
                .
CASE(3):
                .
                .
                .
CASE(6):
                .
                .
                .
```

CASE is declared as an array of statement labels whose bounds are (1:6) and whose 4th and 5th elements are undefined.

Use of elements 4 or 5 produces unpredictable results. It is a poor programming practice to define these arrays such that they have undefined elements.

An array of statement labels cannot be used as an array value, but its elements can be used in any context that permits a statement label.

Names declared by a label prefix in a given block cannot also be declared in a DECLARE statement in the same block, except that the names may be used as the names of members of structures. This means that procedure names, format names, or statement labels cannot also be declared by a DECLARE statement, except as the names of structure members.

Names of external procedures that are part of another program module must be declared by a DECLARE statement if they are to be referenced by the current program module. Only external procedure names can be declared by a DECLARE statement.

DECLARE STATEMENTS

A DECLARE statement declares one or more names by giving the attributes or properties of the named objects. A DECLARE statement is not an executable statement and cannot have a label prefix. It may appear anywhere within a procedure or BEGIN block, except as the THEN clause or ELSE clause of an IF statement or as an on-unit of an ON statement.

It is good programming practice to place all DECLARE statements belonging to a given block at the beginning of that block, rather than mixing them with executable statements. This makes them easy to find when reading the program.

Recommended Forms

The following paragraphs describe the recommended forms of the DECLARE statement.

```
DECLARE name a1 a2 ... an;
```

name is the declared name and a1 through an are attributes. Examples:

```
DECLARE A FIXED BINARY(15);

DECLARE B CHARACTER(10) VARYING STATIC INITIAL('ABC');

DECLARE C DIMENSION(5) FLOAT DECIMAL(7);
```

The last example could also be given without the keyword DIMENSION by writing:

```
DECLARE C(5) FLOAT DECIMAL(7);
```

When several names are to be given the same attributes, the names can be enclosed in parentheses.

```
DECLARE (name-1, name-2, ... ,name-n) a1 a2 ... an;
```

Each name is declared to have all of the attributes a1 through an. Examples:

```
DECLARE (A,B,C) FIXED BINARY(15);

DECLARE (P,Q) POINTER STATIC INITIAL(NULL);

DECLARE (X,Y)(5) FLOAT DECIMAL(7);
```

The last example is equivalent to:

```
DECLARE (X,Y) DIMENSION(5) FLOAT DECIMAL(7);
```

The bounds must be the first attribute in an attribute list if they appear without the keyword DIMENSION.

A structure should be declared using this general form:

```
DECLARE 1  structure_name  a1 a2 ... an,
           2  member_name_1  a1 a2 ... an,
           2  member_name_2  a1 a2 ... an,
                .
                .
                .
           2  member_name_m  a1 a2 ... an;
```

Each member should have a level-number that is one greater than the level-number of its containing structure. The only attributes given to the structure are storage class and DIMENSION. Each member is declared to have all attributes following its name up to the next comma or up to the end of the statement. Example:

```
DECLARE 1  S  STATIC,
              2  A(5)  FLOAT DECIMAL(7),
              2  B  FIXED BINARY(15),
              2  C,
                 3  D  POINTER,
                 3  E  CHARACTER(10)  INITIAL('ABC');
```

S is a static structure with members A, B and C. C is a substructure with members D and E. Initial attributes can be given to elementary members of static structures, but not to structures.

DECLARE Statement General Form

The general form of the DECLARE statement is:

```
DECLARE d1, d2, ... ,dn;
```

where each d is:

```
k name a1 a2 ... an
```

or

```
k (d1, d2, ... ,dn) a1 a2 ... an
```

Where k is an optional level-number, each a is an attribute, and name is a name to be declared. Examples:

```
DECLARE ((A FIXED, B FLOAT) DECIMAL, C BIT) STATIC;

DECLARE 1 S STATIC, 2 (A FIXED, B FLOAT) INITIAL(0);
```

DECLARE statements that contain parenthesized lists of names are called factored declarations. A factored declaration is transformed into a nonfactored declaration by copying the level-number and attribute list of the innermost set of parentheses onto each name contained within the parentheses, removing the parentheses, and repeating this process with the next set of parentheses.

After defactoring, our previous examples are:

```
DECLARE A FIXED DECIMAL STATIC;
DECLARE B FLOAT DECIMAL STATIC;
DECLARE C BIT STATIC;

DECLARE 1  S STATIC,
        2  A FIXED INITIAL(0),
        2  B FLOAT INITIAL(0);
```

Because factored declarations, other than recommended Form 2, are difficult to read, their use is not recommended.

If defactoring produces more than one level-number for a given name, even if the level-numbers are equal, the factored declaration is invalid and causes the compiler to issue an error message.

The specification of duplicate attributes containing more than a simple keyword is also invalid, even when the attributes are exact duplicates.

DECLARATION DEFAULTS

If the attributes specified for a name are incomplete, additional attributes are supplied using these rules:

- If BINARY or DECIMAL is specified without FIXED or FLOAT, FIXED is supplied.
- If FIXED or FLOAT is specified without BINARY or DECIMAL, BINARY is supplied.
- If STATIC is specified without INTERNAL or EXTERNAL, INTERNAL is supplied.
- If EXTERNAL is specified without a storage class and the declared name is not an entry or file constant, STATIC is supplied.
- If the name is the name of a nonparameter variable which is not a member of a structure and no storage class was specified, AUTOMATIC is supplied.
- If BIT or CHARACTER is specified without a length, a length of one is supplied.
- If precision is not specified for arithmetic data, an implementation-defined precision is supplied.
- If a storage class, array bounds, or member's level-number is specified with FILE or ENTRY, VARIABLE is supplied. VARIABLE is also supplied for parameters that have FILE or ENTRY specified.
- If FILE or ENTRY is specified and VARIABLE is not specified and not supplied by the previous rule, EXTERNAL is supplied. In this case, the named object is a constant rather than a variable.

ATTRIBUTE CONSISTENCY

After default attributes have been supplied, each declaration is checked by the compiler for consistency and completeness.

If no data type is specified, the declaration is invalid and incomplete. The compiler issues an error message and supplies a type of FIXED BINARY.

A declaration is inconsistent and invalid if it specifies more than one data type, more than one storage class, or violates any restrictions described for each attribute later in this section.

Valid Data Types:

```
FIXED BINARY(p)
FIXED DECIMAL(p[,q])
FLOAT BINARY(p)
FLOAT DECIMAL(p)
PICTURE
CHARACTER(n) [VARYING]
```

```
BIT(n) [ALIGNED]
POINTER
LABEL
ENTRY [RETURNS][VARIABLE]
FILE [VARIABLE]
BUILTIN
structure
```

Valid Storage Classes:

```
AUTOMATIC
BASED or BASED(pointer-reference)
STATIC INTERNAL [INITIAL]
STATIC EXTERNAL [INITIAL]
DEFINED(reference)
parameter
member of structure [INITIAL only if STATIC]
```

A name declared BUILTIN cannot have any other attribute.

A name declared FILE or ENTRY but without VARIABLE, is a named constant, not a variable, and its scope is external. Note that the presence of a storage class, an array bound, or a member's level-number causes the VARIABLE attribute to be supplied by default to declarations containing ENTRY and FILE.

ATTRIBUTES

The attributes which are permitted in a DECLARE statement are described in this section. They are listed in alphabetical order for easy reference. The discussion of each attribute assumes that the attributes have been made complete by the application of defaults as described in this section.

ALIGNED

ALIGNED is an optional part of the bit-string data type specification. Its presence allows an implementation to align the data on a convenient storage boundary and to use more bits of storage than are specified by the bit-string's declared length. See Section 3 for a discussion of bit-string data.

AUTOMATIC

AUTOMATIC is a storage class attribute and specifies that the declared name is an automatic variable. See Section 4 for a discussion of automatic storage.

BASED or BASED(r)

BASED is a storage class attribute and specifies that the declared name is a based variable. In the second form, r is a reference to a pointer variable or is a pointer-valued function reference and serves as the default or implicit pointer qualifier for unqualified references to the name. See Section 4 for a discussion of based storage.

BINARY(p)

BINARY is part of an arithmetic data type and specifies that the base is binary. The precision (p) may be supplied by this attribute or by the FIXED or FLOAT attribute, but it cannot be specified twice. If not specified, an implementation-defined default precision is supplied (refer to Section 3).

If BINARY is specified without FIXED or FLOAT, FIXED is supplied by default. If either FIXED or FLOAT is specified without BINARY or DECIMAL, BINARY is supplied by default.

When used with FIXED, BINARY specifies integer arithmetic values which contain at least p binary digits.

When used with FLOAT, BINARY specifies floating-point arithmetic values which have a mantissa that contains the equivalent of at least p binary digits.

See Section 3 for a discussion of arithmetic data.

BIT(n)

BIT is part of the bit-string data type specification. The length n is an extent expression or integer constant depending on the storage class of the declared name. A default length of one is supplied if no length is specified. See Section 4 for a discussion of storage classes and Section 3 for a discussion of bit-string data.

BUILTIN

BUILTIN specifies that the declared name is a built-in function. Unless an empty argument list is used in a reference to an argumentless built-in function, the function must be declared with the BUILTIN attribute. BUILTIN may be used to redeclare a built-in function name that was declared as something else in an outer block. The declared name must be one of the names given in Section 10.

CHARACTER(n)

CHARACTER is part of the character-string data type specification. The length n is an extent expression or integer constant depending on the storage class of the declared name. A default length of one is supplied if no length is specified. See Section 4 for a discussion of storage classes and Section 3 for a discussion of character-string data.

DECIMAL(p,q) or DECIMAL(p)

DECIMAL is part of an arithmetic data type and specifies that the base is decimal. The precision (p,q) may be supplied by this attribute or by the FIXED or FLOAT attribute, but cannot be specified twice. If not specified, an implementation-defined default precision is supplied.

If DECIMAL is supplied without FIXED or FLOAT, FIXED is supplied by default. If FIXED or FLOAT is specified without BINARY or DECIMAL, BINARY is supplied by default.

When used with FIXED, DECIMAL specifies fixed-point arithmetic values which contain at least p decimal digits. If q is specified, (p-q) digits are integral digits and q digits are fractional digits. If q is omitted, the values are integers containing at least p decimal digits.

When used with FLOAT, DECIMAL specifies floating-point arithmetic values whose mantissa contains the equivalent of at least p decimal digits. In this case, q cannot be specified.

See Section 3 for a discussion of arithmetic data.

DEFINED(r)

DEFINED is a storage class attribute and specifies that the declared name is a defined variable which shares storage with the variable referenced by r. See Section 4 for a discussion of defined storage.

DIMENSION(b1, b2, ... ,bn)

DIMENSION specifies that the declared name is an array. This attribute is normally written immediately following the variable's name and is written without the keyword DIMENSION. Example:

```
DECLARE X(5) FLOAT;
DECLARE Y DIMENSION(5) FLOAT;
```

Both X and Y are arrays of 5 floating-point values.

Each b represents one dimension and specifies a lower and upper bound for that dimension. Each b has one of these forms:

Form	Definition
*	used only for parameters, an asterisk specifies that both the upper and lower bound of this dimension are to be taken from the corresponding array argument.
lb:hb	lb is an extent expression or optionally signed integer constant depending on the storage class of the array. It specifies the lower bound of this dimension. hb is also either an extent expression or an integer constant depending on the storage class and it specifies the upper bound of this dimension.
hb	hb is an extent expression or an integer constant giving the upper bound of this dimension. The lower bound is assumed to be one.

The DIMENSION attribute cannot be specified for named constants such as FILE or ENTRY. It can only be specified for variables.

DIRECT

DIRECT is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the DIRECT attribute. An attempt to open a file declared as DIRECT with attributes that conflict with the DIRECT attribute will result in a signal of the ERROR condition. DIRECT cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

ENTRY(p1, p2, ... , pn)

ENTRY is a data type attribute. If used without the VARIABLE attribute, it specifies that the declared name is the name of an external procedure. In that case, each p must be a list of attributes that is identical to the attributes specified for the corresponding parameter in that procedure. Example:

```
DECLARE E ENTRY(FIXED BINARY(15), POINTER);
```

This declaration would match an external procedure containing the following statements:

```
E:  PROCEDURE(X,Y);
    DECLARE X FIXED BINARY(15);
    DECLARE Y POINTER;
```

If ENTRY is used with VARIABLE, it specifies that the declared name is a variable whose data type is entry and which can be assigned any procedure name. However at the time when the entry variable is called, it must hold an entry value which designates a PROCEDURE statement whose parameters have attributes which are identical with the corresponding attributes given by p1, p2, etc.

If a PROCEDURE has one or more parameters which are structures, the ENTRY attribute used to declare the procedure name must have a set of attributes for each member of the structure, including all substructures. Example:

```
DECLARE E ENTRY(1, 2 FIXED, 2 FLOAT, POINTER);
```

E is a procedure which has two parameters, the first is a structure having two members, and the second is a pointer. Note that the parameter attributes given in the ENTRY attribute include the level-numbers and attributes of all members as well as the level-number and attributes of the parameter structure.

All string lengths or array bounds given in an ENTRY attribute must be exactly the same as those given in the parameters of the PROCEDURE statement. Programs that violate this rule produce unpredictable results.

An attribute list in an ENTRY attribute is called a parameter descriptor because it describes a parameter.

EXTERNAL

EXTERNAL specifies that the declared name has external scope. This means that all declarations of this name anywhere in the program which also have the EXTERNAL attribute identify the same object. The EXTERNAL attribute can only be used with the STATIC attribute or in the declaration of a file or entry name. If EXTERNAL is specified for a variable and no storage class attribute is specified, STATIC is supplied by default.

FILE

FILE is a data type attribute. If used without the VARIABLE attribute, it specifies that the declared name is a file name which has external scope by default and which has an associated file control block that can be used to perform I/O on files and devices known to the operating system.

If used with VARIABLE, FILE specifies that the declared name is a file variable that can be assigned file values. See Section 2 for a discussion of PL1G I/O.

FIXED(p,q) or FIXED(p)

FIXED is part of an arithmetic data type and specifies that the values have fixed-point scale. The precision (p,q) may be supplied by this attribute or by the BINARY or DECIMAL attribute, but cannot be specified twice. If not specified, an implementation-defined default precision is supplied.

When used with BINARY, FIXED specifies integer arithmetic values that contain at least p binary digits. In this case, q must not be specified.

When used with DECIMAL, FIXED specifies fixed-point arithmetic values that contain at least p decimal digits. If q is specified, p-q digits are integral digits and q digits are fractional digits. If q is omitted, the values are integers containing at least p decimal digits.

See Section 3 for a discussion of arithmetic data.

FLOAT(p)

FLOAT is part of an arithmetic data type and specifies that the values have floating-point scale. The precision (p) may be supplied by this attribute or by the BINARY or DECIMAL attribute, but it cannot be specified twice. If not specified, an implementation-defined default precision is supplied.

If FLOAT is specified without BINARY or DECIMAL, BINARY is supplied by default. If BINARY or DECIMAL is supplied without FIXED or FLOAT, FIXED is supplied by default.

When used with BINARY, FLOAT specifies floating-point arithmetic values whose mantissa contains the equivalent of at least p binary digits.

When used with DECIMAL, FLOAT specifies floating-point arithmetic values whose mantissa contains the equivalent of at least p decimal digits.

See Section 3 for a discussion of arithmetic data.

INITIAL(v1, v2, ... ,vn)

INITIAL specifies the initial value of a static variable or member of a static structure. Each v may be an optionally signed arithmetic constant preceded by an optional parenthesized integer constant iteration factor which indicates that the value is to be used n times; or v may be a character-string or bit-string constant; or v may be a reference to the NULL built-in function and may have an optional parenthesized iteration factor n. Examples:

```
INITIAL(1, 10, 5.2, (5)0)
INITIAL('abc', 'xyz')
INITIAL(NULL)
```

More than one initial value can be specified only if the declared name is an array of m elements. In that case, exactly m initial values must be specified and the values are assigned to the array in row-major order. That is the order in which the elements of an array are allocated storage. See Section 4 for a discussion of array storage.

The INITIAL attribute can be specified only for arithmetic, pictured, string, or pointer variables whose storage class is STATIC or that are members of a STATIC structure. The initial values are converted and assigned to the variables by the compiler and/or loader and must be constants that can be converted to the data type of the variables. NULL may be specified only for pointer variables and must not have been declared the name of anything other than the NULL built-in function.

INPUT

INPUT is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the INPUT attribute. An attempt to open a file declared as INPUT with attributes that conflict with the INPUT attribute will result in a signal of the ERROR condition. INPUT cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

INTERNAL

INTERNAL specifies that the declared name has internal scope. It may be given with any storage class attribute or to parameters, but it has no significance since these variables always have internal scope. If STATIC is used without INTERNAL or EXTERNAL, INTERNAL is supplied by default.

KEYED

KEYED is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the KEYED attribute. An attempt to open a file declared as KEYED with attributes that conflict with the KEYED attribute will result in a signal of the ERROR condition. KEYED cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

LABEL

LABEL is a data type attribute that specifies label values. When used in a declaration of a name, it specifies that the declared name is a label variable. When used in an ENTRY attribute, it specifies that the corresponding parameter is a label variable, and when used in a RETURNS attribute, it specifies that the procedure returns label values.

See Section 3 for a discussion of label values.

OUTPUT

OUTPUT is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the

OUTPUT attribute. An attempt to open a file declared as OUTPUT with attributes that conflict with the OUTPUT attribute will result in a signal of the ERROR condition. OUTPUT cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

PICTURE 'p'

PICTURE is a data type attribute that specifies pictured values. The picture p contains an image of the data and specifies the editing which is to be performed each time that a value is assigned to a pictured variable. It also governs the conversion of pictured values to fixed-point decimal values. See Section 3 for a discussion of pictured data and the picture characters.

POINTER

POINTER is a data type attribute that specifies pointer values. A pointer value is the address of a variable and is discussed in Section 3.

PRINT

PRINT is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the PRINT attribute. An attempt to open a file declared as PRINT with attributes that conflict with the PRINT attribute will result in a signal of the ERROR condition. PRINT cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

RECORD

RECORD is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the RECORD attribute. An attempt to open a file declared as RECORD with attributes that conflict with the RECORD attribute will result in a signal of the ERROR condition. RECORD cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

RETURNS(t)

RETURNS is part of an entry data type specification. It specifies that the entry value designates a procedure which returns a value of data type t, where t is a list of attributes that specify a data type. Examples:

```
DECLARE F ENTRY(FIXED) RETURNS(POINTER);

DECLARE G ENTRY(FLOAT) RETURNS(CHARACTER(32) VARYING);
```

F is declared as the name of a procedure that returns pointer values. G is declared as the name of a procedure that returns varying character-string values whose maximum length is 32 characters.

Any string length given in a RETURNS attribute must be an integer constant. The only attributes that can be given in a RETURNS attribute are data type attributes. DIMENSION or level numbers cannot be given because functions cannot return arrays or structures.

SEQUENTIAL

SEQUENTIAL is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the SEQUENTIAL attribute. An attempt to open a file declared as SEQUENTIAL with attributes that conflict with the SEQUENTIAL attribute will result in a signal of the ERROR condition. SEQUENTIAL cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

STATIC

STATIC is a storage class attribute that specifies that the declared name is a static variable. See Section 4 for a discussion of static storage.

STREAM

STREAM is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the STREAM attribute. An attempt to open a file declared as STREAM with attributes that conflict with the STREAM attribute will result in a signal of the ERROR condition. STREAM cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

UPDATE

UPDATE is an optional part of the declaration of a file constant. It specifies that when the file is opened it will implicitly receive the UPDATE attribute. An attempt to open a file declared as UPDATE with

attributes that conflict with the UPDATE attribute will result in a signal of the ERROR condition. UPDATE cannot be specified for file variables. See Section 2 for a discussion of PL1G I/O.

VARIABLE

VARIABLE is part of a file or entry data type specification. It specifies that the declared name is a file or entry variable rather than a file name or procedure name. See Section 3 which discusses file and entry data.

VARYING

VARYING is part of a character-string data type specification. It specifies that the string values may have any length which does not exceed the declared length. See Section 3 which discusses character-string data.

SECTION 6

REFERENCES

DEFINITIONS

A reference is a name, together with any subscripts, pointer qualifier, or structure names necessary to indicate the purpose of the reference. References to procedures or built-in functions may also contain an argument list. Examples:

```
X

Y(5,K)

P->S.A(K)

F(Z*5+B,SQRT(Z))

Q.NEXT->NODE.FIELD1
```

A reference is associated with a declaration according to the scope of the declared name. The process of associating a reference with a declaration is called resolution of the reference and it occurs during compilation of the program. This section gives the exact rules for resolving references.

SIMPLE AND SUBSCRIPTED REFERENCES

A simple reference is a name without any subscripts, pointer qualifier, etc.

A subscripted reference is a name that has been declared as an array, followed by a parenthesized list of subscript expressions. Each subscript expression must produce an integer fixed-point value that lies within the lower and upper bounds specified for that dimension in the array declaration. The number of subscript expressions must be equal to the number of dimensions. Examples:

```
DECLARE A(5,5) FLOAT;
DECLARE B(10)  FLOAT;
 .
 .
 .
A(K*2,3) = B(1);
```

Both A(K*2,3) and B(1) are subscripted references, K is a simple reference.

STRUCTURE QUALIFIED REFERENCES

A structure qualified reference is a sequence of names written left-to-right in order of increasing level-numbers. The names are separated by periods, and blanks may be written around the periods. The sequence need not include all containing structure names, but it must include sufficient names to make the reference unique.

A structure qualified reference that includes the name of each containing structure from the major structure down to the member is a fully qualified reference. If the name of one or more of the containing structures is omitted, the reference is a partially qualified reference.

Because the names of structure members can be redeclared or reused within the same block, their scopes overlap. If a member's name has been redeclared within the same block, any reference to that member must be qualified by the name of its containing structure. If the containing structure's name has been redeclared within the same block, it must be qualified by the name of its containing structure until an unambiguous reference is created. Example:

```
DECLARE 1  S,
           2  A   FIXED,
           2  B,
              3  A   FLOAT,
              3  C   FLOAT;
```

A reference to A is ambiguous because the scope of A FLOAT overlaps the scope of A FIXED. A structure qualified reference S.A refers to A FIXED. A structure qualified reference to B.A refers to A FLOAT. Also, a structure qualified reference S.B.A. refers to A FLOAT.

In the previous example, B.A is a partially qualified reference and S.B.A is a fully qualified reference, as are S.A, S.B, and S.B.C.

Subscripts can be used anywhere within a structure qualified reference, but it is a good practice to write each set of subscripts immediately following the name that has the corresponding bounds declared for it. Example:

```
DECLARE 1  S(10),
           2  A    FIXED,
           2  B(3) FLOAT,
           2  C(3),
                 3  D    POINTER;
```

A reference to S(K).A and a reference to S.A(K) are equivalent, but S(K).A is preferred. Other recommended references are S(K).B(J) and S(K).C(J).D.

POINTER QUALIFIED REFERENCES

A reference to a based variable may be qualified by a reference to a pointer variable. Example:

```
DECLARE A(10) FLOAT BASED;
DECLARE P POINTER;
```

P->A AND P->A(K) are pointer qualified references to the based variable A. Since a pointer variable may itself be based, multiple qualification is possible. Example:

```
DECLARE 1  NODE  BASED,
           2  NEXT  POINTER,
           2  VALUE FLOAT;

DECLARE HEAD POINTER;
```

HEAD->NODE.NEXT->NODE.VALUE is a pointer qualified reference to NODE.VALUE. It is qualified by NODE.NEXT that is, in turn, qualified by HEAD.

Pointer-valued functions and pointer-valued built-in functions may also be used as pointer qualifiers. Example:

```
DECLARE NEXT_NODE ENTRY RETURNS(POINTER);
```

NEXT_NODE()->NODE.VALUE is a pointer qualified reference to NODE.VALUE whose qualifier is a function reference. Refer to the description of procedure and built-in function references in the following paragraphs.

PROCEDURE REFERENCES

A procedure reference is any reference followed by an argument list consisting of a parenthesized list of expressions separated by commas or followed by an empty argument list (). A procedure reference that returns a value is called a function reference, and one that does not return a value is called a subroutine reference.

If an argument list or empty argument list is given, the referenced name must be declared as an entry either by appearing as a label prefix on a PROCEDURE statement or by a DECLARE statement. Unless the reference is part of a CALL statement, the referenced entry must return a value. If the reference is part of a CALL statement, the entry must not return a value. Therefore, a procedure that returns a value must always be called as a function and a procedure that does not return a value must always be called as a subroutine.

A name declared as an entry is called only when it is referenced with an argument list, or when it is called by a CALL statement. A function that has no parameters must be referenced with an empty argument list.

Except when it is part of a CALL statement, a reference to an entry written without an argument list is a reference to the entry value, rather than a function reference that calls the entry. Example:

```
DECLARE F ENTRY RETURNS(POINTER);
DECLARE G ENTRY(FIXED) RETURNS(FLOAT);
DECLARE V ENTRY VARIABLE RETURNS(POINTER);
  .
  .
  .
V = F;
```

A reference to F() calls F and produces a pointer value. A reference to F is a reference to the entry value of F and can be used as in the example where it is assigned to the entry variable V. A reference to G is also a reference to an entry value and does not call G. G is called by references containing an argument list.

An array of entry variables may be referenced with subscripts and with an argument list. In this case, the argument list follows the subscript list. Example:

```
DECLARE E(5) ENTRY(FIXED) VARIABLE RETURNS(POINTER);
```

E is a reference to the entire array of entry values. E(K)(J) is a pointer valued function reference that calls E(K), passing it the argument J. E(K) is an entry value that may be passed to an entry parameter or assigned to an entry variable. If E had been declared without parameters, an empty argument list would have been used in place of (J) in these examples.

BUILT-IN FUNCTION REFERENCES

A reference to a built-in function always produces the value of that function and is never an entry value. Built-in functions cannot be assigned to entry variables or passed as arguments to entry parameters.

Built-in functions that take arguments do not have to be declared, but built-in functions that take no arguments such as NULL, ONCODE, etc., must either be referenced with an empty argument list () or must be declared with the BUILTIN attribute.

VARIABLE REFERENCES

A reference to a variable may occur in a context that expects a value or it may occur in a context that assigns a value to the variable. If a value is expected, the variable must have previously been assigned a value or must be a static variable declared with the INITIAL attribute. A reference to the value of a variable that has no value produces unpredictable results.

A variable appearing in the left side of an assignment operator or in the list of a GET statement causes assignment to that variable. The ADDR, HBOUND, LBOUND, DIMENSION, and LENGTH built-in functions reference a variable but do not expect a value. (However, LENGTH of a varying string variable does require that the string variable have a value.) A variable passed by-reference to a parameter does not need to have a value if the parameter is not expected to have a value upon entry to its procedure. The record I/O statements copy a variable's storage and do not require the variable to have a value.

REFERENCE RESOLUTION

A fully or partially qualified structure reference is applicable to declarations of structures that include the same hierarchy of names as is used in the structure reference.

A simple or subscripted reference to a name is applicable to any declaration of the name.

A reference is resolved by finding the innermost block that contains any applicable declaration. If no containing block has an applicable declaration, the reference is invalid.

If the block has only one applicable declaration, the reference is resolved to that declaration. If the block has more than one applicable declaration, the reference must be a fully qualified reference to only one declaration in that block; otherwise, the reference is ambiguous and invalid.

Once a block containing an applicable declaration is found, no containing blocks are searched in an attempt to resolve a reference.

The presence of subscripts, arguments, or a pointer qualifier has no effect on the resolution of a reference and cannot make an ambiguous reference unique.

A simple or subscripted reference to a name X is considered to be a fully qualified reference to a nonmember declaration of X. This means that if a member and a nonmember are declared to have the name X in the same block, a reference to X is resolved to the nonmember. The member must be referenced using a structure qualified reference.

SECTION 7

EXPRESSIONS

DEFINITION OF EXPRESSION

An expression consists of operators and operands. An operand may be a constant, a variable reference, a function reference, a built-in function reference, or another expression.

Expression Operators

The operators are:

Operator	Definition
+ - * / **	arithmetic infix
+ -	arithmetic prefix
= ^= > < >= <= ^< ^>	relational
| (or !) &	bit-string infix
^	bit-string prefix
|| (or !!)	concatenate

An infix operator is written between its two operands. A prefix operator is written in front of its operand. Examples:

```
A+B
A*B
A*-B
A>B
A|B
^A
A||B
```

Evaluation of Expressions

The order in which an expression is evaluated is determined by parentheses and by the priority of its operators. The priority of the operators is listed below in descending order, from highest to lowest priority:

```
** ^ prefix + and -
* /
+ -
||
= ^= > < >= <= ^< ^>
&
|
```

Within a parenthesized expression, operators are evaluated in the order of decreasing priority. Example:

```
A**2+B<C
is evaluated as:
((A**2)+B)<C
```

Parentheses are used to alter the order of evaluation. Example:

```
A+B*C
is evaluated as:
A+(B*C)
```

because * has higher priority than +. However, parentheses can be used to force another order. Example:

```
(A+B)*C
```

Operations having the same priority are evaluated from left to right, except for the prefix operators and ** which are evaluated from right to left. Example:

```
A+B+C**-D
is evaluated as:
(A+B)+(C**(-D))
```

If the result of an operator can be determined without evaluating all of its operands, the operands are not necessarily evaluated. A program that depends on all operands being evaluated is invalid and may produce unexpected results when compiled with optimization enabled or when moved to another implementation of PL/I. Likewise, a program that depends on some operands not being evaluated is invalid. Example:

```
IF A = Ø | B/A = 5 THEN ...
```

ARITHMETIC EXPRESSIONS

Operators

The arithmetic operators are:

Operator	Definition
prefix +	plus
prefix -	minus
+	add
-	subtract
*	multiply
/	divide
**	exponentiate

Operand Data Types

The arithmetic operators require arithmetic operands. Pictured values are converted to FIXED DECIMAL values, but other nonarithmetic values must be converted to arithmetic values by one of the conversion built-in functions such as BINARY, DECIMAL, FIXED, or FLOAT.

If the data types of two operands of an infix operator, other than **, differ, the operands are converted to a common arithmetic data type.

If one operand is FIXED and the other is FLOAT, the common type is FLOAT. If one operand is BINARY and the other is DECIMAL, the common base is BINARY if the result is either floating-point or an integer, and the common base is DECIMAL if the result is a fixed-point noninteger. (This latter case is nonstandard and produces an error message from the compiler.)

The precisions of two operands may differ without causing conversion of the operands.

Each arithmetic operator produces a value whose resulting data type is determined by the converted data types of its operands.

Precision Rules

Fixed Point: The fixed-point precision rules effectively allow the result to be formed by aligning the decimal points of the two operands and producing a fixed-point result. The number of digits in the result is always limited by the maximum number of digits allowed by the implementation for the result base, but all fractional digits of the

result are preserved, except for the result of divide. For divide, all integer quotient digits are preserved and as many fractional digits as can be allowed by the implementation are preserved.

Floating Point: Except for exponentiation, floating-point results always have a precision that is the maximum of the precisions of the converted operands.

The following rules give the result's precision for fixed-point values and for the exponential operator.

Prefix: Plus and minus produce a result whose data type is the same as the converted operand.

Addition and Subtraction: Add and subtract of fixed-point values produces a fixed-point result whose base is the common base and whose precision is:

(MIN(N,MAX(P-Q,R-S)+MAX(Q,S)+1),MAX(Q,S))

where N is the implementation's maximum allowed precision for fixed-point values of the result base, (P,Q) is the converted precision of the first operand, and (R,S) is the converted precision of the second operand.

For integer operands, the result precision formula reduces to:

(MIN(N,MAX(P,R)+1))

Multiplication: Multiplication of fixed-point values produces a fixed-point result whose base is the common base and whose precision is:

(MIN(N,P+R+1),Q+S)

where N is the implementation's maximum allowed precision for fixed-point values of the result base, (P,Q) is the converted precision of the first operand, and (R,S) is the converted precision of the second operand.

For integer operands, the result precision formula reduces to:

(MIN(N,P+R+1))

Note

The +1 of the multiplication formula gives more result precision than is needed. This rule derives from the full language where the formula is intended to accommodate complex as well as real fixed-point values.

Division: Divide of fixed-point values is only possible if both operands are fixed decimal. The result is a fixed decimal value of precision:

(N,N-P+Q-S)

where N is the maximum fixed-point decimal precision, (P,Q) is the precision of the first operand, and (R,S) is the precision of the second operand.

For integer operands, this formula reduces to:

(N,N-P)

The formula produces a result quotient that has sufficient precision for all integral digits of the quotient and as many fractional digits as are allowed by the implementation.

The result of division has a large fraction and generally cannot be added or multiplied with another value because alignment of the decimal point with the other value produces a result that is too big for the implementation to support.

The DIVIDE built-in function can be used to divide fixed-point binary values as well as fixed-point decimal values. It provides the programmer with control over the result precision. See Section 1Ø.

Exponentiation: Exponentiation produces a result whose base, scale, and precision depend on the operands and which is determined by one of three cases:

1. If the first operand is fixed-point and the second operand is an integer constant whose value is Y, and if (P+1)*Y-1 does not exceed N, the result is a fixed-point value whose base is that of the first operand and whose precision is:

   ((P+1)*Y-1,Q*Y)

   where N is the maximum precision allowed for fixed-point values of the base of the first operand, (P,Q) is the precision of the first operand, and Y is the value of the second operand.

2. If the second operand is a fixed-point integer value, but Case 1 does not apply, the result is a floating-point value whose base is that of the first operand and whose precision is:

   MIN(N,P)

   where N is the maximum precision allowed by the implementation for floating-point values of the resulting base, and P is the precision of the first operand.

3. If neither Case 1 nor Case 2 applies, the result is a floating-point value having the common base and whose precision is:

MIN(N,MAX(P,R))

where N is the maximum allowed precision for floating-point values having the result base and P is the converted precision of the first operand and R is the converted precision of the second operand.

The result of exponentiation is normally the first operand raised to the power of the second operand. The following are exceptions to this general rule:

if X = Ø and Y>Ø, the result is Ø
if X = Ø and Y<= Ø, ERROR is signalled
if X^= Ø and Y = Ø, the result is 1
if X<Ø and Case 1 does not apply, ERROR is signalled.

X is the value of operand one and Y is the value of operand two.

RELATIONAL EXPRESSIONS

Operators and Results

The relational operators are:

Operator	Definition	
=	equal	
^=	not equal	
>	greater than	
<	less than	
>=	greater than or equal	
<=	less than or equal	
^<	not less than	(equivalent to >=)
^<	not greater than	(equivalent to <=)

The result of a relational operator is always a bit-string of length one that is '1'B if the relationship is true and is otherwise 'Ø'B.

Operand Data Types

If either operand is an arithmetic value or a pictured value, the operands are converted to a common arithmetic type as if they were operands of an add operator. In all other cases, the data types of the two operands must be equivalent.

The ALIGNED, VARYING, RETURNS, VARIABLE and string length attributes are ignored when determining if two data types are equivalent for purposes of these operators. Label, entry, file, and pointer data may only be compared for equality or inequality. Arithmetic and string data may be compared using any relational operator.

Arithmetic and pictured values are compared algebraically.

Character string values are compared from left to-right one character at a time until an inequality is found. The shorter value is extended on the right with blank characters to make it the length of the longer value. Characters are compared using the collating sequence of the computer.

Bit-string values are compared from left to right one bit at a time until an inequality is found. The shorter value is extended with zero bits on its right until it is the length of the longer value.

Pointer values are equal only if they address the same storage location.

Label and entry values are equal only if they designate the same statement and the same stack frame.

File values are equal only if they designate the same file control block.

BIT-STRING EXPRESSIONS

Operators

The bit-string operators are:

Operator	Definition
&	AND
| (or !)	inclusive OR
^	NOT (complement)

Operands

Bit-string operators require bit-string operands. Operands of other data types must be converted to bit-string values using the BIT built-in function.

The infix operators & and | operate on operands of dissimilar length by extending the shorter value to be the length of the longer value by appending zero bits to the right end of the shorter value.

Results

The result of ^ is a bit-string whose bits are the complement of the bits in the operand. (Each Ø bit becomes a 1 and each 1 bit becomes a Ø bit.)

The result of & and | is a bit-string whose length is that of the larger operand. Each bit of the result is given below:

Operand 1	Operand 2	&	|
Ø	Ø	Ø	Ø
Ø	1	Ø	1
1	Ø	Ø	1
1	1	1	1

For example, if X is 'Ø1Ø11'B and Y is '11ØØ1'B, ^X produces '1Ø1ØØ'B, X&Y produces 'Ø1ØØ1'B, and X|Y produces '11Ø11'B. X&'11'B would produce 'Ø1ØØØ'B.

CONCATENATE EXPRESSIONS

Operator

The concatenate operator || (or !!) is used to concatenate two strings producing a string result.

Operands

If both operands are bit-strings, it produces a bit-string result; otherwise, both operands are converted to character-strings, and a character-string result is produced.

Results

The length of the string result is the sum of the lengths of the converted operands. Example:

```
A = 'ABC';
B = 'XYZ';
```

A||B produces 'ABCXYZ', and A||5 produces 'ABC###5' (where # represents a blank character). The blanks in this last example result from the conversion of the fixed-point constant 5 to a character-string. The conversion rule that produces the blanks is explained in Section 8.

SECTION 8

DATA TYPE CONVERSIONS

INTRODUCTION

Each arithmetic, pictured, or string value can be converted to another value whose data type is any other arithmetic, pictured, or string data type. No conversion is possible for data other than arithmetic, pictured, or string data.

Conversions occur as a result of using the assignment, arithmetic, relational, or concatenate operators, from the use of certain built-in functions, from GET/PUT statements, and from some statement options such as KEY and KEYFROM.

Each conversion begins with a source value to be converted and a complete or partial data type called the target data type. The target data type is determined by the context that caused the conversion. For example, the data type of the variable on the left side of the assignment operator provides the target data type for the conversion that is to be performed for that assignment. A partial target data type may result from the use of a conversion built-in function such as FIXED, FLOAT, BINARY, DECIMAL, BIT, or CHARACTER, as well as by other contexts, such as arithmetic operators.

The conversion rules described in Section 7 define the target data type for conversions that result from arithmetic or relational operators. Section 7 also gives the target data type for conversions that result from use of the concatenate operator.

The following kinds of conversion are defined:

Arithmetic to Arithmetic
Arithmetic to Bit-string
Arithmetic to Character-string
Bit-string to Arithmetic
Bit-string to Character-string
Character-string to Arithmetic
Character-string to Bit-string
Format Controlled
Pictured to Arithmetic
Pictured to Bit-string
Pictured to Character-string
Conversion to Pictured

The CEIL function used in many of the conversion rules gives the smallest integer that is greater than or equal to the function's argument. For example, CEIL(1.1) produces the value two.

ARITHMETIC TO ARITHMETIC CONVERSION

This conversion occurs most frequently as a result of using operands of differing arithmetic types in an arithmetic or relational operator. In this case, as well as others, the target base and scale are supplied by the conversion rules of the operator or built-in function, but the target precision is not supplied.

If a target precision is not supplied, it is determined from the data type of the source value and the target base and scale as shown in Table 8-1.

Table 8-1. Target Precisions

Target base and scale, prec r[,s]	Source base and scale, precision p[,q]			
	FIXED BINARY	FIXED DECIMAL	FLOAT BINARY	FLOAT DECIMAL
FIXED BINARY	r = p	r = MIN(CEIL (p*3.32)+1, N)	*	*
FIXED DECIMAL	r = MIN(CEIL (p/3.32)+1, N) s = Ø	r = p s = q	*	*
FLOAT BINARY	r = MIN(p,N)	r = MIN(CEIL (p*3.32),N)	r = p	r = MIN(CEIL (p*3.32),N)
FLOAT DECIMAL	r = MIN(CEIL (p/3.32),N)	r = MIN(p,N)	r = MIN(CEIL (p/3.32),N)	r = p

"*" -- These conversions arise only in cases where the target precision is explicitly known

"N" is the maximum number of digits allowed by the implementation for the target base and scale.

Conversion of a fixed-point decimal value containing a fraction to floating-point or vice versa produces an approximate value.

Conversion of a fixed-point decimal value containing a fraction to a fixed-point value with no fraction or fewer fractional digits results in truncation of excess digits without rounding. Rounding can be performed by using the ROUND, CEIL, FLOOR, or TRUNC built-in functions.

If the target precision does not provide sufficient digits to hold all integral digits of the converted value, the program is in error. This error may or may not be detected by an implementation. If detected, the implementation signals the ERROR condition. If not detected, the program produces unpredictable results. Programs that contain this undetected error and produce "correct" results, may fail when moved to another implementation of PL/I.

The following examples show the target precisions given by Table 8-1 for some common source values and target data types:

Source	Target
FIXED BINARY(15)	FIXED DECIMAL(6,Ø)
FIXED BINARY(31)	FIXED DECIMAL(11,Ø)
FLOAT BINARY(23)	FLOAT DECIMAL(7)
FLOAT BINARY(47)	FLOAT DECIMAL(14)
FIXED DECIMAL(4)	FIXED BINARY(15)
FIXED DECIMAL(7)	FIXED BINARY(25)
FLOAT DECIMAL(6)	FLOAT BINARY(2Ø)
FLOAT DECIMAL(14)	FLOAT BINARY(47)

ARITHMETIC TO BIT-STRING CONVERSION

This conversion converts the absolute value of an arithmetic value to a bit-string and may not produce the result expected by the programmer.

The arithmetic source value is first converted to FIXED BINARY(K) where K is determined by the source data type as shown in the following examples:

Source Data Type	Value of K
FIXED DECIMAL(P,Q)	MIN(N,CEIL((P-Q)*3.32))
FLOAT DECIMAL(P)	MIN(N,CEIL(P*3.32))
FIXED BINARY(P)	P
FLOAT BINARY(P)	MIN(N,P)

The absolute value of the resulting integer is considered to be a bit-string of length K. If the context that caused the conversion supplied a target length, this bit string is extended on the right with zero bits to make it the length of a longer target or excess rightmost bits are truncated to make it the length of a shorter target. The program is in error and may produce unpredictable results if the value of the source is too large to be converted to an integer of precision (K).

For example, a small integer such as 5 whose data type is FIXED BINARY(15) is converted to a bit-string of length 15 whose value is '000000000000101'B. If the target is BIT(16), the result is '0000000000001010'B because padding occurs on the right end. If the target is BIT(5), the result is '00000'B because excess rightmost bits are truncated.

ARITHMETIC TO CHARACTER-STRING CONVERSION

This conversion normally occurs during list-directed stream output, concatenation, or assignment to a character-string.

The arithmetic source value is first converted to a decimal value having the same scale as the source and with a decimal precision determined by the target precision table given in Table 8-1.

The decimal value whose precision is now (p[,q]) is then converted to a character string whose value is given by one of the following cases.

Case 1: If the decimal value is floating-point, the length of the resulting character-string is p+n+4 and its value consists of: n exponent digits preceded by the sign of the exponent, preceded by the letter E, preceded by p-1 digits of the mantissa, preceded by a decimal point, preceded by the most significant digit of the mantissa, preceded by a minus if the value is negative or by a blank if it is not negative. The number of exponent digits depends on the implementation but is constant for a given implementation. Examples:

```
 #0.000000E+00
-7.531000E+02
 #5.499990E-06
```

These values could result from conversion of a FLOAT BINARY(23) value. Seven digits are produced because FLOAT BINARY(23) converts to FLOAT DECIMAL(7) using the target precisions in Table 8-1.

Case 2: If the decimal value is fixed-point with no fraction (q=0), the length of the resulting character-string is p+3 and its value consists of: the p digits of the decimal value with no leading zeroes (the value zero has one zero digit), preceded by a minus if the value is negative, preceded by sufficient blanks to fill the p+3 character result. Examples:

```
########Ø
#######52
#####-5ØØ
```

These values could result from conversion of a FIXED BINARY(15) value. The result is nine characters long because FIXED BINARY(15) converts to FIXED DECIMAL(6) and the result is 6+3 characters long.

Case 3: If the decimal value is fixed-point with a fraction (q^=Ø), the length of the resulting character string is p+3 and its value consists of q fractional digits, preceded by a decimal point, preceded by p-q integral digits with no leading zeros, (fractions and the value zero have one integral zero digit), preceded by a minus if the value is negative, preceded by sufficient blanks to fill the p+3 character result. Examples:

```
####Ø.ØØ
##-5Ø.ØØ
##27.42
####Ø.Ø5
###-Ø.Ø1
```

These values could result from converting a FIXED DECIMAL(5,2) value.

The extra three characters produced for fixed-point values are intended to allow for the case when q = p. In that case, room is needed for a sign, a zero, a decimal point, and p digits.

Programmers who would like to convert arithmetic values to strings that have no leading blanks may use the TRIM function described in Section 1Ø.

BIT-STRING TO ARITHMETIC CONVERSION

Unlike its inverse, bit-string to arithmetic conversion produces reasonable results and permits small bit-strings to be used to hold small positive integers. However, because arithmetic operators require arithmetic operands, the BINARY or DECIMAL built-in function should be used to explicitly convert a bit-string value to a binary or decimal integer value.

The conversion is invalid if the length of the source bit-string exceeds N, where N is the maximum precision allowed by the implementation for fixed-point binary values.

If no target base or scale is given by the context that caused the conversion, FIXED BINARY is supplied by default. If a target precision is not given, the maximum precision allowed for the target base and scale is supplied.

The rightmost bit of the source value is considered to be the units position of a positive binary integer value of precision n, where n is the length of the source bit-string value. The value of that binary integer is then converted to conform to the base, scale, and precision of the target using the normal rules for arithmetic to arithmetic conversion given in Table 8-1. A null bit-string value converts to zero. Examples:

Source	Result
'101'B	5
''B	0
'0000'B	0

BIT-STRING TO CHARACTER-STRING CONVERSION

A bit-string value is converted to a character-string value of the same length as the bit-string. Each bit of the bit-string is converted to a 0 or 1 character in the resulting character-string. A null bit-string is converted to a null character-string.

If the context that caused the conversion gives a target length, the character-string is truncated to conform to a shorter target or is padded on the right with blanks to conform to a longer target. Examples:

Source	Result
'0'B	'0'
''B	''
'1011'B	'1011'

CHARACTER-STRING TO ARITHMETIC CONVERSION

If the context that caused the conversion does not give a target base or scale, FIXED or DECIMAL is supplied by default. If it does not supply a target precision, the maximum precision allowed by the implementation for the target base and scale is supplied.

If the source character-string value is a null string or if it contains all blanks, the result value is zero; otherwise the source string must contain a valid optionally signed constant surrounded by optional blanks. The constant is converted to conform to the data type of the target using the normal rules for arithmetic to arithmetic conversion given in Table 8-1. Examples:

Source	Result
#5E+Ø#	5
#-7###	-7
-4####	-4
##.Ø5#	Ø
######	Ø

These results would be produced by converting a CHARACTER(6) source value to a fixed-point integer result value.

CHARACTER-STRING TO BIT-STRING CONVERSION

A character-string source value that contains any characters other than Ø and 1 cannot be converted to a bit-string value. An attempt to do so results in a signal of the ERROR condition.

A character-string value is converted to a bit-string value of the same length as the character-string. Each character is converted to a Ø or a 1 bit in the resulting bit-string. A null character-string is converted to a null bit-string.

If the context that caused the conversion gives a target length, the bit-string is padded on the right with the required number of zero bits to conform to the longer target, or the rightmost excess bits are truncated to conform to a shorter target. Examples:

Source	Result
''	''B
'Ø1Ø'	'Ø1Ø'B
' '	invalid

FORMAT CONTROLLED CONVERSION

These conversions occur only when a format-list is used by a GET or a PUT statement.

An input conversion occurs when a field of an input line is converted to a result value specified by a data format. That result value is then assigned to a variable given by the list of the GET statement. If the data type of the variable differs from the data type of the result value produced by the format, an additional conversion results from the assignment.

An output conversion occurs when a value given in the list of a PUT statement is converted to a result field by a format.

F-Format

The general form of an F-Format list is:

F(w) or F(w,d)

where w is an integer constant that specifies the width of the field and d is an integer constant that specifies the number of fractional digits in the field.

F-Format Input Conversion: for input conversion, a field of w characters from the input line is converted to a fixed-point decimal value of precision (p,q). If the field contains a decimal point, q is the number of digits following the decimal point; otherwise, q is the value of d or is zero if d is omitted. If the field contains all blanks, the result value is zero and p is the value of MIN(N,w) where N is the maximum precision allowed by the implementation for fixed-point decimal data. If the field does not contain blanks, it must contain an optionally signed fixed-point constant with optional leading or trailing blanks. In that case, p is the precision of the constant. Examples:

Field	Result	Precision
#####	Ø	(5,1)
-7ØØ#	-7Ø.Ø	(3,1)
####Ø	Ø	(1,Ø)
25.##	25	(2,Ø)
#5E+1	invalid	

These results would be produced by an F(5,1) format.

F-Format Output Conversion: for output conversion, an arithmetic or string value from the list of a PUT statement is converted to a fixed-point decimal value that is then rounded and formatted as a character-string of w characters containing a value with d fractional digits.

If d = Ø or is omitted, the source value is converted to a fixed-point decimal value with no fractional digits, and the resulting integer value is placed right justified into the field of w blank characters with leading zeros suppressed. (The value zero has one zero digit.) For negative values, the first significant digit is preceded by a minus. If the value and its sign cannot fit in w characters, the ERROR condition is signalled. Examples:

Value	Result
Ø	###Ø
25	##25
-8	##-8
13.5	##14
17.Ø8	##17
1ØØØ	1ØØØ
-1ØØØ	invalid

These results would be produced by an F(4) format.

If d ^= Ø, the source value is converted to a fixed-point decimal value with d+1 fractional digits. The last fractional digit is rounded by the addition of 5, and it is deleted. The resulting value is placed right-justified into a field of w characters with leading integral zero digits suppressed by blanks (fractional values and zero have one integral zero digit). The leading digit is preceded by a minus if the value is negative. If the value, its decimal point, and its sign cannot fit in w characters, the ERROR condition is signalled. Examples:

Source	Field
Ø	#Ø.ØØ
-1	-1.ØØ
.ØØ5	#Ø.Ø1
.ØØØ5	#Ø.ØØ
1Ø	1Ø.ØØ
-1Ø	invalid

These result would be produced by an F(5,2) format.

E-Format

The general form of an E-Format list is:

E(w) or E(w,d)

where w is an integer constant that specifies the width of the field and d is an integer constant that specifies the number of fractional digits in the field.

E-Format Input Conversion: for input conversion, a field of w characters from the input line is converted to a floating-point decimal value of precision p. If the field contains a decimal point and/or an exponent, the value of d is ignored; otherwise, d indicates that the last d digits in the field are fractional digits. If a field contains all blanks, p is MIN(N,w) where N is the maximum precision allowed by the implementation for floating-point decimal values; otherwise, p is the precision of the constant contained within the field. The field must either be all blank or must contain an optionally signed

fixed-point or floating-point constant preceded or followed by optional blanks. Examples:

Field	Result
######	ØEØ
#-1###	-Ø.1EØ
-25E1Ø	-25E1Ø
#7.41#	7.41EØ
##15Ø#	15EØ

These results would be produced by an E(6,1) format.

E-Format Output Conversion: for output conversion, an arithmetic or string value from a PUT statement's list is converted to a floating-point decimal value and placed right justified into a field of w blanks.

The result field contains n exponent digits, where n is an implementation-defined constant, preceded by the sign of the exponent, preceded by the letter E, preceded by d digits of the mantissa (if d is omitted, it is taken to be p-1), preceded by a decimal point, preceded by the most significant digit of the mantissa, preceded by a minus if the value is negative.

If the value and its sign cannot fit in w characters, the ERROR condition is signalled. Examples:

Value	Format	Field
Ø	E(1Ø,3)	#Ø.ØØØE+ØØ
-15	E(1Ø,3)	-1.5ØØE+Ø1
12345678	E(1Ø,3)	#1.234E+Ø7
7.3E-1Ø	E(1Ø,3)	#7.3ØØE-1Ø
Ø	E(14)	##Ø.ØØØØØØE+ØØ
-25	E(14)	##2.5ØØØØØE+Ø1

The last two examples assume that the precision of the value to be converted is 7, giving a default value of 6 for d.

A-Format

The form of an A-Format list is:

A or A(w)

where w is an integer constant that specifies the width of a field.

A-Format Input Conversion: for input conversion, w must be specified. The result is a character-string containing the next w characters from the input stream file.

A-Format Output Conversion: for output conversion, an arithmetic or string value given by the PUT statement's list is converted to a character-string using the normal rules for conversion to character-string. If w is omitted, it is taken to be the length of this character-string. The string is placed left justified into a field of w blanks.

B-Format

The form of a B-Format list is:

B or B(w)
B1 or B1(w)
B2 or B2(w)
B3 or B3(w)
B4 or B4(w)

where w is an integer constant that specifies the width of a field.

B-Format Input Conversion: for input conversion, w must be specified. The next w characters from the input stream are converted to a bit-string, just as if they had appeared in a bit-string constant that was followed by B, B1, B2, B3, or B4. See Section 3 for a table that gives the valid characters and their translation to bits. If the field of w characters contains an invalid character, the ERROR condition is signalled.

B-Format Output Conversion: for output conversion, an arithmetic or string value given by the PUT statement's list is converted to a bit-string using the normal rules for conversion to bit-strings. The resulting bit-string is then padded on the left with sufficient zero bits to make it a multiple of k bits in length, where k is the 1, 2, 3, or 4 following the B in the format code. The padded bit-string is then converted to a character string of length n, where n is the length of the padded string divided by k. (Each k bits are converted to one character as shown in the table given in Section 3.) If w is omitted, w is the length of this character-string. The character-string is right justified in a field of w blanks. The value of w must be sufficient to hold all of the characters in the string. Examples:

Value	Format	Field
'00'B	B	00
'1'B	B(4)	###1
''B	B(4)	####
'1101'B	B2(2)	31
'110101'B	B3(2)	65
'10011101'B	B4(2)	9D
'10111'B	B2(4)	#113

P-Format

The form of a P-Format list is:

P 'picture'

where picture must be a valid picture as described in Section 3.

P-Format Input Conversion: for input, the next field of w characters is assigned as a pictured value to the variable in the GET statement's list. The number of characters, w, is the number of characters in the picture, excluding any V characters.

P-Format Output Conversion: for output conversion, an arithmetic or string value from the PUT statement's list is converted to a fixed-point decimal value described by the picture. The decimal value is then edited into an output field of w characters as if an assignment had been made to the pictured field. See the discussion at end of this section for a treatment of picture controlled conversion.

PICTURED TO ARITHMETIC CONVERSION

When a pictured value is converted to an arithmetic value, it is first converted to a fixed-point decimal value whose precision is determined by the picture as described in Section 3.

If the context that caused the conversion specified a different data type, the fixed-point value is converted to another value that conforms to the required data type using the normal rules for arithmetic to arithmetic conversion.

PICTURED TO BIT-STRING CONVERSION

Pictured data is converted to bit-string data by first converting the pictured value to a fixed-point decimal value and then converting the decimal value to a bit-string using the rules given in this section for arithmetic to bit-string conversion.

PICTURED TO CHARACTER-STRING CONVERSION

Pictured data is character-string data and is not converted when used in a context that expects character-string data.

CONVERSION TO PICTURED DATA

When an arithmetic or string value is converted to a pictured value, it is first converted to a fixed-point decimal value described by the picture as explained in Section 3. The decimal value is then edited into a character-string of length w, where w is the number of characters in the picture, excluding any V characters.

If the fixed-point decimal value described by the picture is not sufficient to retain all digits to the left of the decimal point, the program is in error and may produce unpredictable results. If fractional digits are lost, they are truncated.

The fixed-point decimal value is edited into the pictured result value under control of the picture edit characters as described in Section 3.

If the fixed-point value is zero and the picture does not contain at least one 9 character, the result is a field of w blanks or w asterisks depending on whether or not an asterisk was used in the picture.

Negative values cannot be edited unless the picture contains at least one sign picture character. Examples:

Value	Picture	Result
5.2	ZZZVZZ	##52Ø
Ø.Ø1	ZZZVZZ	###Ø1
Ø	ZZZ	###
1234	ZZZZV	1234
12345	99999	12345
123	99999	ØØ123
-1Ø5.Ø2	$**,***V.99	invalid
-1Ø5.Ø2	$**,***V.99CR	$***1Ø5.Ø2CR
-75	----V--	#-75ØØ
75	----V--	##75ØØ
-2Ø	-999	-Ø2Ø
2Ø	-999	#Ø2Ø
-275.Ø3	$$$$$V.99-	#$275.Ø3-
25.Ø1	$$$$$V.99-	##$25.Ø1#
-7.5	$$,$$$V.99DB	####$7.5ØDB
Ø	-****V.**	********
5	-****V.**	#***5.ØØ
-75	-****V.**	-**75.ØØ
.75	Z.VZZ	##75
.75	ZV.ZZ	#.75
Ø	ZZ$	###

In the previous picture examples, the pound sign (#) represents a space character.

SECTION 9

STATEMENTS

HOW TO READ THIS SECTION

Each statement is described by giving the general form or syntax of the statement and by describing the effect of executing the statement. Any restrictions imposed by the statement are explained and one or more examples of each statement are shown. Square brackets indicate optional parts of statements and are not part of the punctuation of statements.

Unless a label prefix is required by a statement, it is not shown in the general form of the statement.

ALLOCATE Statement

```
ALLOCATE name SET(reference);
```

Execution of an ALLOCATE statement causes a block of storage of sufficient size to hold the values described by a based variable to be allocated. The address of that block of storage is assigned to the pointer variable referenced in the SET option.

The name must be a simple reference to a based nonmember variable. The SET option must reference a pointer variable. See Section 4 for a discussion of based storage. Example:

```
DECLARE TABLE(1Ø) FLOAT BASED;
DECLARE P POINTER;
   .
   .
   .
ALLOCATE TABLE SET(P);
```

Assignment Statement

```
target = expression;
```

target is either a variable reference or a pseudo-variable as defined later in this section.

Execution of an assignment statement evaluates the target reference and the expression in an unspecified order, converts the expression's value to the data type of the target, and assigns the converted value to the target. The rules for conversion of data types are given in Section 8. Examples:

```
A = B+C;

X(K) = 5;

P->NODE.VALUE = SQRT(X(J));

SUBSTR(S,I,J) = 'ABC';
```

String Assignment

Assignments to bit-string targets result in padding of the source value on the right with sufficient zero bits to make it the length of the target.

Assignments to nonvarying character-string targets result in padding of the converted source value on the right with sufficient blanks to make it the length of the target.

Assignment to a bit or character-string target that is shorter than the converted source value causes truncation of the rightmost bits or characters of the source value.

Assignment to a varying character-string target causes the target to obtain a new current length of the newly assigned value.

Assignment Rules

If the target is a reference to an entire array or structure, the expression must also be a reference to an entire array or structure that is identical in size, shape, and component data types to the target. In this case, the array or structure referenced by the expression is copied into the storage of the target. No other form of assignment involving arrays or structures is permitted.

If the target is a reference to a character-string variable and the expression is also a reference to a character-string variable, the target and the expression variables must not partially overlap each other in storage such that the target begins to the right of the source. This restriction also applies when either or both the target and the expression are SUBSTR references. This restriction does not forbid A = A; or A = A!!B; or A = SUBSTR(A,2,3); but does forbid SUBSTR(A,2,3) = A; and any equivalent assignment.

Because the target and the expression are evaluated in a unspecified order, functions or on-units called during these evaluations must not assign to subscripts or pointers used within the target reference. Likewise, the storage of the target must not be freed by the execution of a function or on-unit called during evaluation of the expression.

Programs that depend on the order of evaluation may fail when moved to other implementations of PL/I.

PAGENO Pseudo-variable Assignment Statement

```
PAGENO(f) = e;
```

f must be a reference to a file name or to a file variable that has been assigned a file value.

The file control block must be open and must describe a STREAM OUTPUT PRINT file.

The expression e is evaluated and converted to a binary integer value of implementation defined precision and is assigned as the current page number of the file control block identified by f. This assignment does not cause any additional pages to be output on the file, it simply changes the current page number that serves as the value of the PAGENO built-in function. Example:

```
DECLARE F FILE;
   .
   .
   .
PAGENO(F) = 1Ø;
```

STRING Pseudo-variable Assignment Statement

```
STRING(r) = e;
```

r is a reference to a string, array, or structure variable. The variable identified by r must be suitable for string overlay storage sharing as described in Section 4.

The converted value of the expression e is assigned to r as if r were a nonvarying string variable whose length is the total number of characters or bits in the variable identified by r. Example:

```
DECLARE A(5) CHARACTER(1);
   .
   .
   .
STRING(A) = 'ABCDE';
```

After this assignment, A(1) has the value of 'A', A(2) has the value of 'B', etc.

SUBSTR Pseudo-variable Assignment Statement

```
SUBSTR(r,i,j) = e;
  or
SUBSTR(r,i) = e;
```

r must be a reference to a character or bit-string variable and i and j must be fixed-point integer-valued expressions. If r identifies a varying character-string, it must have a current value. In that case, let w be the current length of r. If r identifies a nonvarying string, w is the declared length of the string variable. If j is omitted, it is assumed to equal w - i + 1. The following restrictions on i, j, and w must be satisfied or the program is in error and produces unpredictable results:

```
1<=i<=w + 1
i+j-1<=w
```

These restrictions ensure either that i is the index of a character or bit within the string and that a substring of j characters or bits beginning with the ith character or bit can be accessed without exceeding the length of the string, or that i is one greater than the length of the string and j is zero. If j is omitted, the substring begins with the ith character or bit and extends to the end of the string. No other portion of the target string is affected by the assignment.

The substring described by i and j is assigned the converted value of the expression e as if the substring were a nonvarying string. Example:

```
DECLARE S CHARACTER(10);
   .
   .
   .
SUBSTR(S,3,4) = 'ABC';
```

After the assignment, the third character of S is 'A', the fourth is 'B', the fifth is 'C', and the sixth is blank. The other characters of S are unchanged by the assignment.

UNSPEC Pseudo-variable Assignment Statement

```
UNSPEC(r) = e;
```

where r is a reference to any variable except an array or structure.

The expression e is evaluated and converted to an implementation-defined bit-string that is copied into the storage of r. Subsequent use of r is invalid unless the bit-string is a valid value for that variable.

Use of the UNSPEC pseudo-variable is a poor programming practice that makes a program dependent on the implementation and prevents the program from being moved to another implementation of PL/I. Example:

```
UNSPEC(C) = '0'B;
```

After this assignment, the storage of C contains zero bits.

BEGIN Statement

```
BEGIN;
```

Execution of a BEGIN statement causes a block activation of the BEGIN block defined by the BEGIN statement. The block activation is terminated when the corresponding END statement is executed, or it may be terminated by execution of a GOTO or RETURN statement. (RETURN returns from the containing procedure, not from the BEGIN block.)

A BEGIN statement defines a BEGIN block and is an executable statement that may appear as a THEN clause or ELSE clause of an IF statement, as an on-unit of an ON statement, or as a simple statement anywhere in a procedure.

Unless a BEGIN block is an on-unit or contains declarations whose scope must be limited by the BEGIN block, it should not be used because it consumes more computer time than does a simple DO group that could be used in its place. Examples:

```
BEGIN;
   .
   .
   .
END;

ON ERROR
   BEGIN;
      .
      .
      .
   END;
```

CALL Statement

```
CALL reference;
```

Execution of a CALL statement creates a block activation of the procedure identified by the reference. The reference must be an entry name, entry variable, or entry-valued function. The procedure designated by that entry value is called and passed any arguments given with the reference. The procedure thus activated must not have a RETURNS option and must have the same number of parameters as the reference has arguments. A procedure with no arguments may be called using a reference with an empty argument list () or no argument list.

Each argument in the argument list is evaluated and, if necessary, converted to the data type of the corresponding parameter. See Section 4 for a discussion of argument passing and parameters.

The order in which arguments and other components of the reference in a CALL statement are evaluated is undefined and may vary from one implementation of PL/I to the next. Examples:

```
CALL E(A,B,5+X);
CALL F;
CALL G();
```

CLOSE Statement

```
CLOSE FILE(reference);
```

Execution of a CLOSE statement closes the file control block identified by the reference. The reference must identify a file name, file variable, or file-valued function.

Closing a closed file has no effect and is not an error.

Once closed, the file control block may be reopened and given different file attributes and may be used to designate a different operating system file or device. Examples:

```
CLOSE FILE(F);
CLOSE FILE(G(K));
```

DECLARE Statement

```
DECLARE variable attributes;
      or
DECLARE (variable-list) attributes;
```

Execution of a DECLARE statement has no effect. A DECLARE statement cannot be used as a THEN clause or ELSE clause of an IF statement or as the on-unit of an ON statement. A DECLARE statement cannot have a label prefix.

The purpose of a DECLARE statement is to declare the names of variables, external procedures, and files. It is discussed more fully in Section 5. Examples:

```
DECLARE (A,B,C) FIXED BINARY(15);
DECLARE F FILE;
```

DELETE Statement

```
DELETE FILE(reference) [KEY(expression)];
```

The FILE and KEY options may be given in any order.

Execution of a DELETE statement deletes a record from a KEYED SEQUENTIAL UPDATE file. The reference must identify a file value whose associated file control block has been opened with the KEYED, SEQUENTIAL, and UPDATE attributes. The key option is specified, the key expression is evaluated and converted to a varying character-string whose length is implementation defined. If a record with that key does not exist, the KEY condition is signalled. Return from a KEY condition on-unit resumes execution with the statement following the DELETE statement.

If the KEY clause is omitted, the current record of the file is deleted. Examples:

```
DELETE FILE(F) KEY(25);
DELETE KEY(C||'.OLD') FILE(G);
```

DO Statement

There are four kinds of DO statements: simple-do, do-while, do-repeat, and iterative-do.

All DO statements define a group of statements called a do-group that begins with the DO statement and ends with the corresponding END statement.

A do-group is executed a variable number of times under the control of its DO statement as explained later in this section.

A DO statement cannot be used as an on-unit, but can appear anywhere within a procedure or BEGIN block, including a THEN or ELSE clause of an IF statement.

In the discussions that follow, we assume that the do-group does not transfer control out of the group or skip statements within the group when we say that the statements of the group are executed once, twice, n times, etc. However, any do-group may contain IF statements, other do-groups, RETURN, or GOTO statements that alter the order of execution.

If control is transferred out of non-simple do-group, control cannot be transferred back into the group. Likewise, control cannot initially be transferred into a non-simple do-group except by execution of its DO statement.

Simple-do Statement

```
DO;
```

Execution of a simple DO statement causes the statements in the do-group to be executed once. Control may be transferred into a simple do-group by a GOTO statement.

Do-while Statement

```
DO WHILE(expression);
```

Execution of a do-while statement causes the expression to be evaluated to produce a value b that must be a bit-string of length one. If b is true, the statements of the group are executed and when the corresponding END statement is executed, control is transferred back to reevaluate the expression and test the new value of b. If b is false, the statements in the group are not executed and execution resumes with the statement following the END statement. Example:

```
K = 1;
DO WHILE(K<=10);
   .
   .
   .
K = K+1;
END;
```

This do-group is executed 10 times.

Do-repeat Statement

```
DO index = start REPEAT next [WHILE(test)];
```

index is a variable reference, and start, next, and test are expressions.

Execution of a do-repeat causes the start expression to be evaluated and assigned to the index variable. If a WHILE option is given, the test expression is then evaluated to produce a value b that must be a bit-string of length one. If b is false, execution resumes with the statement following the corresponding END statement. If b is true or if a WHILE option is not given, the statements of the group are executed. When control reaches the END statement, the next expression contained in the REPEAT option is evaluated and assigned to the index variable. Any WHILE option is again evaluated and if it is true or not present, the group is executed again.

Index must be a reference to a variable whose data type makes it a suitable target for assignment of both start and next. Both start and next may be expressions of any data type, provided that they both can be assigned to index.

Note

Unless a WHILE is given, the loop repeats indefinitely.

The index reference is not completely reevaluated each time that a next value is assigned to it. The original values of any subscripts, pointer qualifiers, or string lengths, used in index are used in all subsequent assignments of next values. The next and test expressions are completely re-evaluated each time.

The index variable must not be an array or structure, but it may be a reference to an element of an array or a member of a structure. Example:

```
DECLARE (HEAD,P) POINTER;
DECLARE  1  NODE  BASED,
            2  VALUE FLOAT,
            2  NEXT  POINTER;

DO P = HEAD REPEAT(P->NODE.NEXT) WHILE(P^=NULL);
   .
   .
   .
END;
```

This do-group is repeatedly executed for each non-null pointer in the chain of nodes rooted in the pointer HEAD.

Iterative-do Statement

DO index = start [TO finish] [BY increment] [WHILE(test)];

index is a variable reference, and start, finish, increment, and test are expressions.

Execution of an iterative-do causes the start, finish, and increment expressions as well as the index reference to be evaluated in an unspecified order. Once evaluated, the finish and increment values are used throughout execution of the group and are not reevaluated. The subscripts, pointers, etc., in the index reference are also not reevaluated.

The start value is converted, if necessary, to the data type of the index variable and the converted value is assigned to the index variable.

If a WHILE option is given, the test expression is evaluated to produce a value b that must be a bit-string of length one. If b is false, execution resumes with the statement following the END statement.

If b is true or a WHILE option is not given, the value of the index variable is compared to the finish value. If the increment is positive or zero and if the index value is greater than the finish value, execution resumes with the statement following the END statement.

If the increment is negative and the index value is less than the finish value, execution resumes with the statement following the END statement.

If none of the above conditions caused execution of the group to terminate, the statements of the group are executed, the increment value is added to the index variable and any WHILE test is reevaluated. The new values of b and index are then used to determine if the group should be executed again.

The TO and BY options may be given any order followed by the WHILE option. Either the TO or the BY option may be omitted. If the BY option is omitted, a default increment of one is supplied. If the TO is omitted, index is not compared against any limit and the group repeats indefinitely unless stopped by the WHILE.

Note

If both the TO option and the WHILE option are omitted, the do-group ~~repeats indefinitely~~ executes once (PTU84).

The start, finish, and increment expressions must produce integer fixed-point values. The index must be a reference to a fixed-point integer variable. Examples:

```
DO K = 1 TO 1Ø;
   .
   .   /* EXECUTES 1Ø TIMES */
   .
END;

DO K = 1Ø TO 1 BY -1;
   .
   .   /* EXECUTES 1Ø TIMES */
   .
END;

DO K = 1 TO 1Ø WHILE(B<Ø);
   .
   .
   .
END;

DO A(J) = B+1 TO N BY -M WHILE(T);
   .
   .
   .
END;
```

All executions of the last example use the same element of A as the index, even if J has been altered by the execution of the group. They also all use the initial values of B+1, N and -M. However, T is repeatedly evaluated.

END Statement

```
END [name];
```

The END statement terminates a DO group, BEGIN block, or PROCEDURE block.

Execution of an END statement that closes a do-group may cause the group to be repeated depending on the DO statement that heads the group. See the description of DO in the preceding paragraphs.

Execution of an END statement that closes a procedure is valid only if the PROCEDURE statement does not contain a RETURNS option. The END statement terminates the current block activation and returns control to the statement following the CALL statement that called this procedure.

Execution of an END statement that closes a BEGIN block causes termination of the current block activation and resumes execution of the previous block activation with the statement following the END statement.

If the BEGIN block is an on-unit, control returns to the source of the signal. This is not possible if the on-unit has been established for the ERROR condition. An attempt to return to the source of the ERROR condition terminates execution of the program and produces an execution-time error message.

An END statement may have a label prefix and can be referenced by a GOTO statement, including GOTO statements contained within the do-group or block closed by the END statement.

If a name is given, it must be the same name that occurs in the label prefix of the corresponding DO, BEGIN or PROCEDURE statement. If it is not the same name, the compiler issues an error message.

An END statement cannot appear as a THEN clause, ELSE clause, or on-unit. Examples:

```
P:  PROCEDURE;
    DO K=1 TO 10;
       .
       .
       .
    END;
    END P;
```

FORMAT Statement

```
name:  FORMAT(format-list);
```

Execution of a FORMAT statement has no effect unless it occurs as a consequence of the execution of a GET or PUT statement.

A FORMAT statement must have an unsubscripted label prefix that serves as the format name. A format name is not a statement label and cannot be used by a GOTO statement.

A format-list is used during the execution of a GET or PUT statement to control the transmission of data to or from a stream I/O file. It consists of a list of format-items separated by commas. Each format-item may have an optional repetition factor consisting of an integer constant k whose value is 0<=k<=255. Each format-item is one of the following:

```
(format-list)
A(w) or A
B(w) or B
B1(w) or B1
B2(w) or B2
B3(w) or B3
B4(w) or B4
E(w) or E(w,d)
F(w) or F(w,d)
P 'picture'
R(name)
COLUMN(n) or COL(n)
LINE(n)
PAGE
SKIP or SKIP(n)
TAB or TAB(n)
X(n)
```

Each w, d, or n is an integer constant whose value k must be 0<=k<=255, unless further restricted by the definition of a particular format-item.

Each time control passes to a format-list, all format-items between the last used format-item and the next data format-item are evaluated. The next data format-item is then used to control the conversion of the data being transmitted to or from the stream file.

If control reaches the end of a format-list in a FORMAT statement, control returns to the R-format that transferred control to the FORMAT statement.

If control reaches the end of a format-list in a GET or PUT statement, and one or more values remain to be transmitted in the statement's I/O list, control transfers to the beginning of the format-list.

The following sections describe the effect of each format-item.

Nested Format-lists

If a parenthesized format-list is used as a format-item, it is used as many times as specified by its repeat count. Each time that control reaches the end of the parenthesized format-list, the repeat count is reduced by one and the entire list is repeated until the count is zero. Example:

```
F:  FORMAT(F(10,4),2(A(5),E(14,3)),SKIP);
```

This example is equivalent to:

```
F: FORMAT(F(10,4),A(5),E(14,3),A(5),E(14,3),SKIP);
```

Data Formats

The data formats are A, B, B1, B2, B3, B4, E, F, and P. They control the transmission and conversion of a value to or from a stream file. Each data format causes w characters to be read from an input stream or written to an output stream, where w is the field width given in the format or calculated during the conversion and given by the conversion rules in Section 8.

If fewer than w characters remain in an input line, any characters remaining on that line are read and a new line is obtained. Additional characters are then read from the new line so that a total of w characters are read.

If fewer than w characters remain in an output line, as many characters as fit are written on that line and a new line is begun. Any remaining characters are written onto the new line so that a total of w characters are written.

See Section 8 for a detailed description of how each data format controls conversion of a value.

R(format-name)

An R-format transfers control to the format-list of the FORMAT statement whose name appears in the R-format. The effect of this transfer is as if the format-list of the FORMAT statement were called as a subroutine, i.e. when the remote format-list is exhausted, control returns to the R-format and is passed to the format-item following the R-format. Example:

```
F:  FORMAT(A,X(3));
PUT EDIT(P,Q) (R(F),E(14,3));
```

The value of P is transmitted by the A format contained in the FORMAT statement. Transmission of Q finishes the remote format list by evaluating the X control format and returns to use the E format to transmit the value of Q.

COLUMN(n) or COL(n)

A COLUMN format puts blanks into an output stream or skips characters of an input stream so that the next character is be read or written into column n of a line.

If the current output line contains exactly n-1 characters, no output is performed and the next output begins in column n of the current line.

If the current output line contains less than n-1 characters and n is less or equal to the line size, sufficient blanks are placed into the current line to cause the next output to begin in column n.

If the current output line contains less than n-1 characters and n is greater than the line size of this file, the current line is written and a new line is begun. The next output begins in column 1 of the new line.

If the current output line already contains more than n characters, it is written to the stream file and a new line is begun. If n is greater than the line size of this file, no blanks are placed into the new line; otherwise, n-1 blanks are placed into the new line causing the next output to begin in column n of the line.

If the current input line is positioned such that the next character to be read is located in column n, no characters are skipped.

If the current input line is positioned such that the current column is less than column n and n does not exceed the size of the current line, sufficient characters of the line are skipped so that the next input occurs from column n.

If the current input line is positioned such that the current column is less than column n, but n exceeds the size of the current line, or if the current column is greater than column n, a new line is read. If n exceeds the size of the new line, no characters are skipped and the next input occurs from column 1; otherwise, n-1 characters are skipped causing the next input to occur from column n.

The value n must be greater than zero.

COL(n)	COLUMN	LINENO	NEW COLUMN	NEW LINENO
25	25	1	25	1
24	25	1	24	2
26	25	1	26	1
30	25	1	1	2

These examples assume a linesize S that is 26<=S<30.

LINE(n)

A LINE format can only be used to control output to a stream file that has been opened with the PRINT attribute and that, consequently, has a page size. It positions the stream file to a specified line relative to the top of a page.

If the current line number is less than n and n does not exceed the page size, sufficient lines are written to the output so that the current line is line number n.

If the current line number is greater than n or if n exceeds the page size, the remainder of the page is filled with empty lines and the ENDPAGE condition is signalled, unless the current line number already exceeds the page size. In that case, a new page is begun without signalling the ENDPAGE condition.

If n equals the current line and the current column is one, no output occurs. If n equals the current line, but the current column is not one, the remainder of the page is filled with blank lines and the ENDPAGE condition is signalled, unless the current line number already exceeds the page size. In that case, a new page is begun without signalling the ENDPAGE condition.

The value n must be greater than zero.

PAGE

A PAGE format can only be used to control output to a stream file that has been opened with the PRINT attribute. It positions the stream to the top of a new page and thereby resets the line number to one and increases the page number by one.

SKIP(n) or SKIP

A SKIP format applied to an input stream skips the rest of the current input line and n-1 subsequent lines, i.e., it skips over n line boundaries.

A SKIP format applied to an output stream writes the current line and n-1 empty lines, i.e., it writes n line boundaries.

If n is omitted, a value of one is supplied. The value of n must be greater than zero.

If the output stream file has been opened with the PRINT attribute and if the total number of lines written as a result of a SKIP would exceed the page size, the current line is written, followed by sufficient empty lines to fill the page, and the ENDPAGE condition is signalled.

TAB(n) or TAB

A TAB format can only be used to control output to a stream file opened with the PRINT attribute. It causes sufficient blanks to be placed into the current line to position it to the nth tab stop relative to the current column of the line. The relative positions of tabs are implementation defined and may or may not be at constant intervals from each other.

If the current column is a tab stop, a TAB produces sufficient blanks to cause the next output to begin at the next tab stop.

If n tab stops do not remain on the current line, the current line is written and a new line begun. If the first tab stop is not in column 1, sufficient blanks are placed into this line to position it to the first tab stop.

The line size must not be less than the first tab stop if TAB formats are used. If n is omitted, a default value of one is supplied. The value of n must be greater than zero.

X(n)

The X format writes n blanks to the current output stream or skips n characters in the current input stream.

If less than n characters remain in the current line, the remaining characters are skipped on input or written as blanks on output, and additional characters are skipped or written as blanks on the next line so that a total of n characters are written or skipped.

The value of n must be greater than zero.

FREE Statement

```
FREE reference;
```

Execution of a FREE statement frees the block of storage that is identified by the reference. The reference must be a reference to a based variable and the pointer qualifier used explicitly or implicitly by the reference must point to a block of storage allocated by execution of an ALLOCATE statement.

The reference must not be subscripted and must identify a nonmember based variable whose size, shape and component data types are the same as were used when the storage was allocated.

Once freed, a block of storage must not be referenced. Any pointers that address the block are invalid and must not be used. Violation of this rule produces unpredictable results. Example:

```
DECLARE TABLE(100) FLOAT BASED;
DECLARE P POINTER;
   .
   .
   .
ALLOCATE TABLE SET(P);
   .
   .
   .
FREE P->TABLE;
```

The FREE statement frees the storage block allocated by the ALLOCATE statement.

GET Statement

```
GET [FILE(f)] [SKIP[(n)]] LIST(input-list);
   or
GET [FILE(f)] [SKIP[(n)]] EDIT(input-list) (format-list);
```

input-list is input-item [,input-item]...

An input-item is either a:

```
variable-reference
  or
(input-list iterative-do)
```

format-list is defined in this section in the discussion of the FORMAT statement.

The FILE, SKIP, and LIST options or the FILE, SKIP, and EDIT options may be given in any order, but a format-list is part of the EDIT option and must immediately follow the input-list. If SKIP is given without (n), a value of one is supplied. If FILE is not given, FILE(SYSIN) is supplied by default.

Note

No comma separates the iterative-do from its associated input-list.

An input-list consisting of one iterative-do has two sets of parentheses. Example:

```
GET FILE(F) LIST((A(K),B(K) DO K = 1 TO 10));
```

The FILE option must reference a file value whose associated file control block has either been opened as a STREAM input file, or is closed. If closed, it is opened by the GET statement (see Section 2 and Section 9) and given the STREAM and INPUT attributes.

The expression in the SKIP option must produce a positive fixed-point integer value n. The SKIP option skips across n line boundaries and resets the current column to one.

After any SKIP option has been evaluated, the input-list is evaluated together with any format-list.

The input-list is evaluated from left-to-right. Each variable reference may be either an array reference, structure reference, or scalar variable reference.

A scalar variable reference causes one value to be transmitted from the input stream and, if EDIT is specified, it uses one data format. An array variable causes n values to be transmitted where n is the number of elements in the array. If EDIT is specified, it uses n data formats. Values are transmitted to the array in row-major order as defined under ARRAYS in Section 3. A structure variable causes all members of the structure and members of all contained substructures to receive a value. The values are transmitted in left-to-right order. If EDIT is specified, each value requires a data-format.

Only arithmetic, pictured, and string values can be transmitted by a GET statement.

A parenthesized input-list containing an iterative-do transmits values under control of the iterative-do as if it were a do-group. Examples:

```
DECLARE A(1Ø) FLOAT;
DECLARE (B,C) FLOAT;
   .
   .
   .
GET FILE(F) LIST(A,B,C);
GET FILE(F) EDIT(A(K),K,B(K)) (3 E(14,6));
GET FILE(F) LIST(B,(A(K) DO K = 1 TO 5),C);
```

The first GET statement transmits 1Ø values to the array A and then transmits values to B and C in that order.

The second GET statement transmits a value to A(K), transmits a value to K, and then transmits a value to B(K) using the new value of K as a subscript. All three values are transmitted using the same E format.

The last example transmits a value to B, transmits values to A(1), A(2), A(3), A(4), and A(5), and to C in that order.

The number of lines read by a GET statement is determined by the size of the list, the SKIP option, and any control formats given in the format-list. However, unless control items or SKIP forces new lines to be read, transmission begins with the current position of the current line and uses as many lines as are necessary to satisfy the input-list.

List-directed Input

A LIST option specifies list-directed input. It causes each value expected by the input-list to be read from the stream file without the use of a format-item.

For list-directed input, the stream file must contain a sequence of fields each of which is followed by a blank or a comma. Excess blanks may occur between fields.

A field whose value is to be assigned to an arithmetic, pictured, or bit-string variable x must contain an optionally signed constant of the same form as could appear on the right side of an assignment statement whose target was X.

A field whose value is to be assigned to a character-string variable begins with the next nonblank character in the stream. If that character is a quote ('), the field is treated as a character-string constant and ends with the matching quote (which must be followed by a blank or comma); otherwise, it ends with the character immediately before the next blank or comma. If the entire field is a valid character-string constant, its value is assigned to the list variable; otherwise, all characters of the field are assigned without conversion to the list variable.

An empty field is a field terminated by a comma and contains only blanks or contains no characters. Examples:

```
,5#
##,#,,
```

The last line contains two empty fields(assuming that the previous field was terminated by a blank). .

An empty field causes no assignment to its corresponding list variable.

Note that all fields, including the last field in a file, must be followed by a blank or a comma.

A field may be split across two or more lines, i.e. a line boundary does not terminate a field. However, to facilitate input from certain devices such as on-line terminals, an implementation may insert a blank at the end of each line read from a terminal and thereby prevent a field from continuing onto another line.

Multiple blanks following a field are scanned when the field is read and the stream is positioned at the next nonblank or end of the line, whichever occurs first. Examples:

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

If the stream contains:

```
#52#1.Ø7E+5#ABC
X#7
```

A is assigned 52, B is assigned 1.Ø7E+5, and C is assigned 'ABCX'. The position of the file after the second GET statement is executed is column 3. If the second line only contained two columns, the stream would be positioned at the end of the line such that any subsequent input would read a new line or detect end of file if no lines remained. If the 7 on the second line had been a comma, it would be ignored by the next GET list operation, but would be read if the next operation was GET EDIT.

Edit-directed Input

An EDIT option specifies edit-directed input. It uses a format-list to control scanning of the input stream and conversion of fields. Each field is first converted under control of a data format and the converted value is then assigned to the corresponding list variable. An additional conversion may occur as part of that assignment.

See Section 8 for a description of each data format conversion and see the discussion of the FORMAT statement in this section for a description of format-list evaluation. Example:

```
GET FILE(F) EDIT(A,B,C) (E(14,3),SKIP,F(1Ø));
GET FILE(F) EDIT(D) (A(1Ø));
```

A is assigned the field converted by E(14,3), a new line is read by SKIP, B is assigned the field converted by F(1Ø), and C is assigned the next field converted by E(14,3). The SKIP is not evaluated a second time because no variables remain in the list. D receives a value from the field converted by A(1Ø). That field is the 1Ø characters that immediately follow the second field converted by E(14,3) and assigned to C.

GO TO Statement

```
GO TO reference;
```

or

```
GOTO reference;
```

Execution of a GOTO statement transfers control to the statement designated by the reference. The reference must identify a label value.

If the label value is a label variable or label-valued function, its value must designate a statement in a currently active block and the value must also designate a stack frame belonging to a block activation of that block.

If the statement to which control is transferred is not within the current block, the current block activation is terminated as are all previous block activations back to the block containing the statement. That block activation is made current, and control is transferred to the statement.

See Section 3 for a more complete discussion of label values. Examples:

```
GOTO L;
GOTO CASE(K);
GO TO L;
```

IF Statement

```
IF expression THEN clause [ELSE clause]
```

clause is either a BEGIN block, do-group, or statement other than END, PROCEDURE, DECLARE, or FORMAT.

Execution of an IF statement causes the expression to be evaluated to produce a value b, that must be a bit-string of length one. If b is '1'B, the THEN clause is executed; otherwise, the ELSE clause is executed if specified.

When execution of a THEN clause is complete, the statement following the IF statement is executed. If b is 'Ø'B and no ELSE is given, the statement following the IF statement is executed.

When IF statements are used as a THEN or ELSE clause of another IF, any ELSE is always matched with the nearest preceding THEN. Example:

```
IF A>B
   THEN IF C>D
           THEN X = 5;
           ELSE X = 1Ø;
```

If an ELSE is to be associated with the first THEN, an ELSE clause (or an empty ELSE) must be written with the second THEN:

```
IF A>B
   THEN IF C>D
        THEN X = 5;
        ELSE;
   ELSE X = 10;
```

A statement appearing as a THEN or ELSE clause must not have a label prefix. However, a BEGIN block or do-group used as a clause may contain labeled statements.

Null Statement

```
;
```

Execution of a null statement has no effect. Null statements are used to provide null THEN or ELSE clauses in IF statements, null on-units in ON statements, or to allow multiple label prefixes. Examples:

```
A:;
B:;
C: ON ENDPAGE (F);
```

The ON statement has a null on-unit and is preceded by two labeled null statements effectively giving the ON statement three labels. However, these labels do not compare equal because they designate different statements.

ON Statement

```
ON condition-name on-unit;
```

on-unit is a BEGIN block or a statement other than PROCEDURE, DO, END, DECLARE, FORMAT, or RETURN. Condition-name is ERROR, ENDFILE(f), ENDPAGE(f), or KEY(f).

Execution of an ON statement establishes the on-unit as if it were a procedure that is to be called when the condition is signalled. It does not execute the on-unit.

If an on-unit for this condition has already been established in the current block activation, it is replaced by this on-unit.

An on-unit remains established until it is replaced by another, until it is reverted by a REVERT statement, or until the block activation in which it was established is terminated.

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, etc., until an on-unit for the condition is found. If no on-units exists, a default on-unit is invoked. The default on-unit for KEY or ENDFILE signals the ERROR condition. The default on-unit for ENDPAGE puts a new page. The default on-unit 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 choose to let its caller's on-unit handle the condition. Any on-unit established by a block is reverted when the block returns to its caller or is otherwise terminated.

A signal causes an on-unit to be called just as if it were a procedure that had no parameters. The block activation resulting from this call is terminated when the on-unit executes a GOTO or executes its END statement. In the latter case, control returns to the source of the signal. On-units for the ERROR condition cannot return to the source of the signal. An attempt to do so produces an execution-time error message and terminates program execution.

A GOTO executed within an on-unit and transferring control out of the on-unit terminates the block activation of the on-unit and any block activations back to, but not including, the activation of the block to which control is transferred.

See Section 2 for a discussion of block activation and for a discussion of exception handling.

The I/O conditions of ENDFILE, ENDPAGE, and KEY are uniquely established for each file control block and are always written with a reference f that must identify a file value. The condition thus referenced is effectively qualified by the file control block associated with f.

This means that ENDFILE(F) and ENDFILE(G) are different conditions if F and G identify different file control blocks, but are the same condition if F and G identify the same file control block.

Just before these conditions are signalled, the value of ONFILE is set to the file-id of the file control block for which the condition is being signalled.

An on-unit must not have a label prefix.

ENDFILE(f) Condition

The ENDFILE condition is signalled when an attempt is made to read past the end of a file. The end-of-file status of the file control block remains set so that subsequent reads also cause the signal to occur.

Returning from the on-unit transfers control to the statement following the GET or READ statement.

The default on-unit writes an error message onto an implementation-defined error file and signals the ERROR condition.

ENDPAGE(f) Condition

The ENDPAGE condition is signalled when the line to be written has a line number that is one greater than the page size of the STREAM OUTPUT PRINT file identified by f.

If an on-unit returns without writing a new page, the line number of the file increases indefinitely and the ENDPAGE condition is not signalled by subsequent output. However, if a new page is written by the on-unit or at any time later, it resets the line number and allows ENDPAGE to be signalled the next time that the line number is one greater than the page size.

Returning from the on-unit returns to the point where the signal occurred and any additional output is then written to the stream.

The default on-unit puts a new page and returns.

ERROR Condition

The ERROR condition is signalled whenever the implementation detects an error. Just prior to signalling the condition, the value of the ONCODE built-in function is set to an implementation-defined integer value that serves as an error code indicating which error occurred.

Returning from the on-unit produces an error message and terminates program execution.

The default on-unit writes an error message on an implementation-defined error file and terminates program execution.

KEY(f) Condition

The KEY condition is signalled when a READ, REWRITE, or DELETE statement containing a KEY option cannot find a record with the specified key. It is also signalled by a WRITE statement whose KEYFROM option specifies a key of an existing record.

Just before the condition is signalled, the key value is assigned as the value to be returned by the ONKEY built-in function.

Returning from the on-unit resumes execution with the statement following the record I/O statement.

The default on-unit writes an error message on an implementation-defined error file and signals the ERROR condition.

OPEN Statement

```
OPEN FILE(f) [TITLE(s)] [LINESIZE(n)] [PAGESIZE(m)] [STREAM]
  [RECORD] [INPUT] [OUTPUT] [UPDATE] [PRINT] [KEYED] [SEQUENTIAL]
  [DIRECT];
```

The options and attributes may be specified in any order.

Execution of an OPEN statement causes the file control block identified by f to be opened with the line size, page size, and attributes specified in the OPEN statement. If specified, n and m must be expressions that produce fixed-point positive integer values.

The FILE option must be specified and f must be a reference that produces a file value. If the file control block identified by f is already open, the OPEN statement is ignored, even if its attributes disagree with those of the file control block.

If no TITLE option is specified, the file control block is connected to a file or device known to the operating system using the file-id as the title. (The file-id of a file control block is the name of the file constant that owns the control block as explained under Files in Section 2.) If a TITLE is specified, s must be an expression that produces a character-string value. The character-string is used to identify a file or device known to the operating system. Because the title is used by the operating system, it has an implementation-defined format that may include such information as the name of the file, its record size, or accessing mode, etc.

The attributes specified in an OPEN statement may be an incomplete set. The attribute set is made complete by first adding attributes given in the declaration of the file, then supplying implied attributes and then supplying default attributes. See Section 2 and Section 5. The final set of attributes must be one of the consistent sets defined below.

Stream-attributes are:

```
STREAM INPUT
STREAM OUTPUT [PRINT]
```

Record-attributes are:

```
RECORD INPUT SEQUENTIAL [KEYED]
RECORD INPUT DIRECT KEYED
RECORD OUTPUT SEQUENTIAL [KEYED]
RECORD OUTPUT DIRECT KEYED
RECORD UPDATE SEQUENTIAL KEYED
RECORD UPDATE DIRECT KEYED
```

Examples

```
OPEN FILE(F) STREAM INPUT;
OPEN FILE(F) TITLE ('MASTER_FILE.NEW') UPDATE;
OPEN FILE(G) LINESIZE(80) PAGESIZE(60) STREAM OUTPUT PRINT;
```

PROCEDURE Statement

```
name: PROCEDURE[(parameter-list)] [RETURNS(t)] [RECURSIVE] [OPTIONS
(MAIN)];
```

The parameter-list is:

```
name [,name]...
```

and t is a list of data type attributes.

The RETURNS, RECURSIVE and OPTIONS (MAIN) options may be specified in any order.

Execution of a PROCEDURE statement as a consequence of the normal flow of control from the previous statement has no effect and execution resumes with the statement following the procedure's END statement.

A PROCEDURE statement defines a procedure that consists of a block of statements closed by the procedure's corresponding END statement. The PROCEDURE statement must have a label prefix that declares the name of the procedure. That declaration is established in the block which contains the PROCEDURE statement, and consequently makes the name known in that block as well as all contained blocks. The procedure's END statement should contain the name of the procedure so that it can be easily identified as the closing END statement.

Each name in the parameter-list must be declared within the procedure and must not be a member of a structure and must not be declared with a storage class attribute.

Every call to the procedure must be made with an argument list containing the same number of arguments as there are parameters in the parameter-list, and each argument must be capable of being passed either by-reference or by-value to its corresponding parameter. See Section 4 for a discussion of argument passing.

If a RETURNS option is not specified, the procedure must always be called by a CALL statement, and it must not contain any RETURN statements that specify a return value.

If a RETURNS option is specified, t must be a set of data type attributes that specify a scalar value. All values returned by the procedure are converted to this data type prior to being returned as the function value of the procedure. All RETURN statements must specify a return value and the procedure must not execute its own END statement. All calls to the procedure must result from the evaluation of a function reference.

If the procedure is called recursively, the RECURSIVE option must be specified.

If OPTIONS (MAIN) is specified, a return from this procedure effectively acts as a STOP statement, closing all PL1G files and terminating program execution.

Procedures may contain other procedures and BEGIN blocks as well as any other statements. Examples:

```
P:  PROCEDURE(A,B) RETURNS(FIXED BINARY(15));
    DECLARE(A,B) FLOAT;
     .
     .
     .
    END P;
Q:  PROCEDURE(X) RECURSIVE;
    DECLARE X CHARACTER(*);
     .
     .
     .
    CALL Q('HELLO');
     .
     .
     .
   END Q;
```

PUT Statement

```
PUT [FILE(f)] [SKIP[(n)]] [LINE(m)] [PAGE] [list];
```

list is:

```
LIST(output-list)
  or
EDIT(output-list) (format-list)
```

and output-list is:

```
output-item [,output-item]...
```

and output-item is:

```
expression
  or
(output-list iterative do)
```

If specified, n and m must be expressions that produce fixed-point integer values, and f must be a reference that produces a file value. The discussion of the FORMAT statement in this section gives the definition of a format-list.

The options may be given in any order, but the format-list is part of the EDIT option and must always immediately follow the output-list.

If SKIP is given without n, a value of one is supplied by default.

If the FILE option is omitted, FILE(SYSPRINT) is supplied by default. When SYSPRINT is opened, it acquires the PRINT attribute by default.

Note

No comma separates an iterative-do from its output-list.

An output list containing one iterative-do has two sets of parentheses. Example:

```
PUT FILE(F) LIST((A(K),B(K) DO K = 1 TO 10));
```

The FILE option must reference a file value whose associated file control block has either been opened as a STREAM OUTPUT file or is closed. If closed, it is opened by the PUT statement and given the STREAM and OUTPUT attributes, as well as the PRINT attribute if the file-id is SYSPRINT.

The expression in the SKIP option must produce a positive fixed-point integer value n. The option writes n lines beginning with the current line so that any subsequent output begins on a new line.

Either SKIP or LINE, but not both, may be specified. The LINE option is evaluated as if it were a LINE format as described in the discussion of the FORMAT statement in this section and positions the stream file to a specific line relative to the top of a page. Subsequent output begins on that line.

The PAGE option puts a new page so that subsequent output begins on line one of that page. If PAGE and LINE are given, PAGE is evaluated first.

SKIP, PAGE, and LINE are always evaluated prior to writing any output produced by the statement.

If the output from a statement does not share its last line with subsequent output, two statements must be used. Example:

```
PUT FILE(F) LIST(A,B,C);  PUT FILE(F) SKIP;
```

If a PUT statement is used to produce all subsequent lines, a SKIP can be used in the same statement, but it produces an empty line at the beginning of the output stream. Example:

```
DO K = 1 TO 1000;
  .
  .
  .
PUT FILE(F) SKIP LIST(A,B,C);
  .
  .
  .
END;
```

After any SKIP option has been evaluated, the output-list is evaluated together with any format-list.

The output-list is evaluated from left-to-right. Each expression may be either an array reference, structure reference, or scalar-valued expression.

A scalar value causes one value to be transmitted to the output stream and, if EDIT is specified, uses one data format. An array variable causes n values to be transmitted where n is the number of elements in the array. If EDIT is specified, it uses n data formats. Values are transmitted from the array in row-major order as defined in Section 3. A structure variable causes all members of the structure and members of all contained substructures to transmit a value. The values are transmitted in left-to-right order. If EDIT is specified, each value requires a data-format.

Only arithmetic, pictured, or string values can be transmitted by a PUT statement.

A parenthesized output-list containing an iterative-do transmits values under control of the iterative-do, as if it were a do-group. Examples:

```
DECLARE A(10) FLOAT;
DECLARE (B,C) FLOAT;
   .
   .
   .
PUT FILE(F) LIST(A,B,C);
PUT FILE(F) EDIT(A(K),C) (E(14,6));
PUT FILE(F) LIST(B,(A(K) DO K = 1 TO 5), C);
```

The first PUT statement transmits 10 values from the array A and then transmits values from B and C in that order.

The second PUT statement transmits a value from A(K), and then transmits a value from C. Both values are transmitted using the same E format.

The last example transmits a value from B, transmit values from A(1), A(2), A(3), A(4), A(5), and from C in that order.

The number of lines written by a PUT statement is determined by the number of values specified by the OUTPUT LIST as well as by the SKIP, LINE, and PAGE options, and any control formats given in the format-list. However, unless control items or options force new lines to be written, transmission begins with the current position of the current line and uses as many lines as are necessary to satisfy the output-list.

List-directed Output

A LIST option specifies list-directed output. It causes each value specified by the output-list to be written to the stream file without the use of a format-item.

Each value to be output is converted to a character-string value using the normal rules for conversion to character-string given in Section 8.

If the original output value is a bit-string, the resulting character-string is enclosed in quotes and a B is appended to its right end.

If the original output value is a character-string or pictured and the file control block does not have the PRINT attribute, each contained quote is replaced by two quotes and the entire string is enclosed by quotes.

The possibly modified character-string is placed into the output stream followed by a single blank. If the file control block has the PRINT attribute, the value is followed by sufficient blanks to ensure that the next output begins in the next tab stop. At least one blank always separates fields.

If a character-string does not fit on a line, a new line is begun and the string is written on that line and subsequent lines if necessary. Examples:

```
PUT FILE(F) LIST(52,1.Ø7E+5);
PUT FILE(F) LIST('ABCX');
```

The previous examples would produce the following output:

```
###52##1.Ø7E+Ø5#
'ABCX'#
```

The value 52 converts to ###52 and is output followed by a single blank, 1.Ø7E+5 converts to #1.Ø7E+Ø5 and is followed by a single blank, 'ABCX' converts to ABCX and has quotes attached to each end. The resulting value does not fit on a line of size 2Ø so it is written on the next line followed by a blank. If the file control block had the PRINT attribute, the last value would be output without quotes and each value would be followed by sufficient blanks to ensure that the next value would begin at the next tab stop.

Edit-directed Output

An EDIT option specifies edit-directed output. It uses a format-list to control the position of the output stream and the conversion of the values specified by the output-list. Each output value is converted under control of a data format and the resulting characters are written to the output stream.

See Section 8 for a description of each data format conversion and see the discussion of the FORMAT statement in this section for a description of format-list evaluation. Examples:

```
PUT FILE(F) EDIT(A,B,C) (E(14,3),SKIP,F(1Ø));
PUT FILE(F) EDIT(D) (A);
```

A is converted under control of E(14,3) and 14 characters are written to the output stream, a new line is begun by SKIP, B is converted under control of F(1Ø) and 1Ø characters are written to the output stream, C is then converted under control of E(14,3) and written to the output stream. The SKIP is not evaluated a second time because no more values remain to be output from the output-list. The second statement converts D to a character-string whose length is determined by the normal rules for conversion to character-string. The resulting value is written to the same line as C, unless C happened to just fill a line. In that case, D begins on the next line.

READ Statement

```
READ FILE(f) INTO(v) [KEY(k)] [KEYTO(r)];
```

The FILE, INTO, KEYTO, and KEY options may be written in any order.

KEYTO must not be specified if KEY is specified.

Execution of a READ statement reads a record from a record file by copying the record into the storage of the variable referenced by the INTO option. If the file control block has been opened with KEYED SEQUENTIAL, the KEY option may be specified. If the file control block has been opened with DIRECT, the KEY option must be specified.

If specified, the KEY option must be an expression whose value can be converted to a character-string of implementation defined length. Its value must be the key-value of a record in the keyed file identified by the file control block associated with f.

An attempt to read using a key value that does not identify a record in f results in a signal of the KEY condition.

The FILE option must contain a reference f that produces a file value. The file control block associated with f must either have been previously opened with RECORD INPUT or must be closed. If closed, it is opened by the READ statement and given INPUT RECORD SEQUENTIAL.

If specified, the KEYTO option must reference a varying character string variable whose maximum length is implementation-defined. The file must be a keyed file. The key value of the record is assigned to r.

If a KEY option is given, the file control block must have been previously opened with the KEYED attribute. The presence of the KEY option does not cause implicit opening to produce the KEYED attribute.

Regardless of how the file control block is opened, it must have either INPUT or UPDATE as well as RECORD.

If a KEY option is given, the file is positioned to read the record identified by the key value; otherwise, it is positioned to read the current record of a SEQUENTIAL file. After reading the record, the current position is advanced to the next record if the file is SEQUENTIAL.

The record is a copy of storage and must have been produced by a WRITE or REWRITE statement whose FROM option identified a variable whose size, shape, and component data types were identical to those of the variable identified by the INTO option. Variables used in INTO and FROM options must not be unaligned bit-strings or structures consisting entirely of unaligned bit-strings. Violation of these rules produces unpredictable results and may or may not cause the ERROR condition to be signalled. Examples:

```
READ FILE(F) INTO(X);
READ FILE(G) INTO(Y) KEY(N+1);
```

RETURN Statement

```
RETURN [(result)];
```

result must be an expression whose values can be converted to the data type specified in the RETURNS option of the containing procedure.

Execution of a RETURN statement terminates the current block activation and returns control to the calling block.

If a result is specified, the containing procedure must have a RETURNS option and must have been called by a function reference. In this case, the result expression is evaluated, converted to the data type specified by the RETURNS options of the PROCEDURE statement, and returned as the function value.

If a result is not specified, the containing procedure must not have a RETURNS option and be called only by a CALL statement.

A RETURN statement executed in a BEGIN block returns from the activation of the containing procedure block and terminates any BEGIN blocks that contain the RETURN statement.

A BEGIN block that is an on-unit must not contain a RETURN statement. Some examples of the RETURN statement are:

```
RETURN;
RETURN(A+B);
RETURN('STRING RESULT');
RETURN('1'B);
```

REVERT Statement

```
REVERT condition-name;
```

condition-name may be ERROR, ENDFILE(f), ENDPAGE(f), or KEY(f).

Execution of a REVERT statement reverts the on-unit established within the current block activation by a previously executed ON statement. If no on-unit for the condition is established in the current block activation, the REVERT statement has no effect.

The reference f given in an I/O condition name must produce a file value. The I/O condition is qualified by the file control block I/O associated with f. This means that REVERT ENDPAGE(F); and REVERT ENDPAGE(G); revert different on-units if F and G designate different file control blocks; otherwise, they revert the same on-unit. Examples:

```
REVERT ERROR;
REVERT ENDPAGE(F);
```

REWRITE Statement

```
REWRITE FILE(f) FROM(v) [KEY(k)];
```

The FILE, FROM, and KEY options may be given in any order.

Execution of a REWRITE statement replaces a record in a KEYED UPDATE file.

Because the set of file attributes that would be supplied as a result of an implicit file opening caused by a REWRITE statement does not include KEYED, the FILE option must reference a file value whose associated file control block has been opened with the KEYED and UPDATE attributes.

The KEY option, if specified, must contain an expression whose value can be converted to character-strings of an implementation-defined length. If the KEY option is omitted, the file must have been opened with the KEYED and SEQUENTIAL attributes. In this case, the current record of the file will be replaced.

The FROM option must contain a variable reference. The storage of that variable is copied as the new record. The FROM option must not contain a variable that is an unaligned bit-string or a structure that consists entirely of unaligned bit-strings. Examples:

```
REWRITE FILE(F) FROM(X) KEY(N+1);
REWRITE FILE(F) FROM(Y(K)) KEY('ABC');
```

SIGNAL Statement

```
SIGNAL condition-name;
```

condition-name may be: ERROR, ENDFILE(f), ENDPAGE(f), or KEY(f).

Execution of a SIGNAL statement signals the specified condition. It is normally used during program debugging to test on-units.

A signal from a SIGNAL statement or from the PL/I implementation calls the most recently established on-unit for the specified condition.

See Section 2 for a discussion of exception handling. Examples:

```
SIGNAL ERROR;
SIGNAL ENDFILE(F);
```

STOP Statement

```
STOP;
```

Execution of a STOP statement closes all open files and terminates program execution.

WRITE Statement

```
WRITE FILE(f) FROM(v) [KEYFROM(k)];
```

The FILE, FROM, and KEYFROM options may be written in any order.

Execution of a WRITE statement writes a record into the file identified by the FILE option by copying the storage of the variable referenced by the FROM option. The variable must not be an unaligned bit-string or a structure consisting entirely of unaligned bit-strings.

If the file control block associated with f has been opened with the KEYED and SEQUENTIAL attributes, the KEYFROM option may be specified. If it has been opened with the DIRECT attribute, the KEYFROM option must be specified.

If specified, the KEYFROM option must contain an expression whose value can be converted to a character-string of implementation-defined length that serves as the key-value of the new record. If a record with this key-value already exists, the KEY condition is signalled.

The FILE option must contain a reference f that produces a file value. The file control block associated with f must have been either previously opened with RECORD OUTPUT, or it must be closed. If closed, it is opened by the WRITE statement and given OUTPUT RECORD SEQUENTIAL.

If a KEYFROM option is given, the file control block must have been previously opened with the KEYED attribute. The presence of a KEYFROM option does not supply the KEYED attribute in an implicit file opening. Examples:

```
WRITE FILE(F) FROM(X);
WRITE FILE(G) FROM(Y) KEYFROM(N+1);
```

SECTION 10

BUILT-IN FUNCTIONS

SUMMARY

The built-in functions may be grouped into the following classes:

- Arithmetic Built-In Functions:

  ABS CEIL DIVIDE EXP FLOOR
  LOG LOG2 LOG10 MAX MIN
  MOD ROUND SIGN SQRT TRUNC

- Trigonometric Built-In Functions:

  ACOS ASIN ATAN ATAND ATANH
  COS COSD COSH SIN SIND
  SINH TAN TAND TANH

- String Built-In Functions:

  BOOL COLLATE COPY INDEX LENGTH
  STRING SUBSTR TRANSLATE VALID
  VERIFY

- Conversion Built-In Functions:

  BINARY BIT BYTE CHARACTER DECIMAL
  FIXED FLOAT RANK

- Condition Built-In Functions:

  ONCODE ONFILE ONKEY ONLOC

- Miscellaneous Built-In Functions:

  ADDR DATE DIMENSION HBOUND LBOUND
  LINENO NULL PAGENO UNSPEC TIME

FUNCTION DESCRIPTIONS

▶ ABS(X)

The result is the absolute value of X and has the same data type as X. X must be an arithmetic value.

▶ ACOS(X)

The result is the arccosine of X and has the same data type as X. X must be a floating-point value. The result is expressed in radians.

▶ ADDR(X)

The result is a pointer to the storage referenced by X. X must not be a reference to a parameter whose corresponding argument was passed by-value. X must not be a parameter whose corresponding argument is an array that is a member of a dimensioned structure because the storage of such an array is fragmented and cannot be accessed by a pointer and a based variable. On many implementations, X must not be an unaligned bit-string or a structure consisting entirely of unaligned bit-strings.

▶ ASIN(X)

The result is the arcsine of X and has the same data type as X. X must be a floating-point value. The result is expressed in radians.

▶ ATAN(X)

The result is the arctangent of X and has the same data type as X. X must be a floating-point value. The result is expressed in radians.

▶ ATAN(X,Y)

The result is the angle in radians whose tangent is X/Y. Both X and Y must be floating-point values. The result is the common type and maximum precision of X and Y.

▶ ATAND(X)

The result is the arctangent of X and has the same data type as X. X must be a floating-point value. The result is expressed in degrees.

▶ ATAND(X,Y)

The result is the angle in degrees whose tangent is X/Y. Both X and Y must be floating-point values. The result is the common type and maximum precision of X and Y.

▶ ATANH(X)

The result is the hyperbolic arctangent of X and has the same data type as X. X must be a floating-point value.

▶ BINARY(X) or BINARY(X,P)

X may be an arithmetic or string value. If X is a fixed-point decimal with a non-zero scale factor then P must be given. P is an integer constant indicating the precision of the result.

If X is floating-point, the result is a float binary value; otherwise, the result is a fixed-point binary value. If P is omitted, the result has a precision that is determined by the rules for type conversion given in Section 8.

▶ BIT(S) or BIT(S,L)

S may be an arithmetic or string value. L must be a positive fixed-point integer. If L is given, S is converted to a bit-string of length L. Otherwise, S is converted to a bit-string whose length is determined by the rules for type conversion given Section 8.

▶ BOOL(X,Y,Z)

X and Y must be bit-string values. Z must be a bit-string constant, four bits long.

The result is a bit-string whose length is the maximum of the lengths of X and Y.

If X and Y are null strings, the result is a null string. If X is not the same length as Y, the shorter string is extended on the right with zero bits until X and Y are the same length. The bits values within Z are m1, m2, m3, m4 from left to right respectively.

The i-th bit value of the result is set to one of the values m1, m2, m3, m4 depending on the i-th bit value of X and Y according to the following table:

X(i)	Y(i)	RESULT(i)
0	0	m1
0	1	m2
1	0	m3
1	1	m4

For example, the result of BOOL('11ØØ11Ø','Ø1Ø1',Ø11Ø') is '1ØØ111Ø'.

▶ BYTE(X)

A PL1G extension. The result is a single character selected by SUBSTR(COLLATE(),X+1,1). X must be a fixed-point binary integer value. Note that the resulting character is formed by taking the rightmost bits of X as a byte. On computers that use the ASCII character set with the high order bit of each byte set to '1', the definition of this function is SUBSTR(COLLATE(),X-127,1).

▶ CEIL(X)

The result is the smallest integer greater than or equal to X and has the same data type as X.

If X is a floating-point value, the precision of the result is the precision of X; otherwise, the precision of the result is (MIN(N,MAX(p-q+1,1)),Ø). N is the maximum precision allowed for fixed-point values of the result type. Examples:

The result of CEIL(-3.1) is -3.

The result of CEIL(3.1) is 4.

The result of CEIL(Ø) is Ø.

▶ CHARACTER(S) or CHARACTER(S,L)

S may be an arithmetic or string value, and L must be a positive fixed-point integer value.

If L is given, S is converted to a character-string of length L; otherwise, S is converted to a character-string whose length is determined by the rules for type conversion given in Section 8.

▶ COLLATE() or COLLATE

The result is a character-string of implementation-defined length that consists of the set of characters in the computer's character set in ascending order.

▶ COPY(X,Y)

X must be a string value and Y must be a positive fixed-point integer value. The result is obtained by concatenating Y occurrences of X.

▶ COS(X)

The result is the cosine of the angle X expressed in radians and has the same data type as X. X must be a floating-point value.

▶ COSD(X)

The result is a value that is the cosine of the angle X expressed in degrees, and has the same data type as X. X must be a floating-point value.

▶ COSH(X)

The result is a value that is the the hyperbolic cosine of the angle X expressed in radians, and has the same data type as X. X must be a floating-point value.

▶ DATE() or DATE

The result is a character-string that represents the system date of the form YYMMDD where YY, MM, and DD are in the ranges ØØ:99, Ø1:12, and Ø1:31 and represent the year, month, and day respectively.

▶ DECIMAL(X) or DECIMAL(X,P) or DECIMAL(X,P,Q)

X may be an arithmetic or string value. P is an integer constant indicating the precision of the result. If P alone is given, Q is assumed to be zero. Q must be an integer constant indicating the scale of the result.

If X is floating-point, the result is a float decimal value; otherwise, the result is a fixed decimal value. In the former case, Q must be omitted. If Q is omitted and the result is fixed-point, the result is an integer of precision P. If P is omitted, the result has a precision determined by the rules for type conversion given in Section 8.

▶ DIMENSION(X,N)

X must be an array variable. N must be an integer constant indicating the N-th dimension of X.

The result is a fixed binary integer giving the number of elements in the N-th dimension of X.

The precision of the result is implementation-defined. X must have at least N dimensions, and N must be greater than zero. Example:

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,1Ø:1Ø,4:7);
 .
 .
 .
R=DIM(A,1);        /*R=3*/
 .
 .
```

▶ DIVIDE(X,Y,P) or DIVIDE(X,Y,P,Q)

X is an arithmetic value, Y is an arithmetic value, P is an integer constant indicating the precision of the result, Q is an integer constant indicating the scale of the result.

The result is X/Y and has the common data type of X and Y.

The precision of the result is (P,Q) or (P).

If Y = Ø, the program is in error and the results of continued execution are undefined.

▶ EXP(X)

The result is the value of the base of the natural logarithm e raised to the power of X: e**X, and has the same data type as X. X must be a floating-point value.

▶ FIXED(X,P) or FIXED(X,P,Q)

X may be an arithmetic or string value, P is an integer constant indicating the precision of the result. Q must be an integer constant indicating the scale of the result.

The result is X converted to a fixed-point arithmetic value according to rules for type conversion given in Section 8.

▶ FLOAT(X,P)

X may be an arithmetic or string value, P is an integer constant indicating the precision of the result.

The result is X converted to a floating-point arithmetic value according to the rules for type conversion given in Section 8.

▶ FLOOR(X)

The result is the largest integer that is less than or equal to X.

If X is a floating-point value, the precision of the result is the precision of X; otherwise, the precision of the result is (MIN(N,MAX(p-q+1,1)),Ø). Examples:

The result of FLOOR(3.125) is 3.

The result of FLOOR(-3.125) is -4.

The result of FLOOR(Ø) is Ø.

▶ HBOUND(X,N)

X must be an array variable, N must be an integer constant indicating the N-th dimension of X.

The result is a fixed binary integer giving the upper bound of the N-th dimension of X.

The precision of the result is implementation-defined.

X must have at least N dimensions, and N must be greater than zero. Example:

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,-10:10,4:7);
 .
 .
 .
R = HBOUND(A,1);              /*R = 5*/
 .
 .
 .
R = HBOUND(A,2);              /*R = 2*/
 .
 .
```

▶ INDEX(S,C)

S and C must be character-strings, or S and C must be bit-strings.

The function searches a string S for a specified substring C and returns a fixed binary integer value indicating the position of C within S.

The result precision is implementation-defined.

If either S or C is a null string, the result is zero. If the substring C is not contained within S, the result is zero; otherwise, the result is an integer indicating the position within S of the leftmost character or bit of the substring C.

For example, the result of INDEX('abcdefg','def') is 4.

▶ LBOUND(X,N)

X must be an array variable, N must be an integer constant indicating the N-th dimension of X.

The result is a fixed binary integer value giving the lower bound of the N-th dimension of X.

The precision of the result is implementation-defined.

X must have at least N dimensions, and N must be greater than zero. Example:

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,-10:10,4:7);
 .
 .
 .
R = LBOUND(A,1);          /*R = 3*/
 .
 .
 .
R = LBOUND(A,2);          /*R = 1*/
 .
 .
```

▶ LENGTH(S)

S is either a character- or bit-string.

The result is a fixed binary integer giving the number of characters or bits in the string S.

The precision of the result is implementation-defined. The null string has length zero.

▶ LINENO(X)

The result is a fixed binary integer giving the line number of the file control block identified by X. X must be a file value.

The precision of the result is implementation-defined.

X must identify an open file control block with the PRINT attribute.

▶ LOG(X)

The result is the natural logarithm of X, and has the same data type as X. X must be a floating-point value greater than zero.

▶ LOG10(X)

The result is the logarithm of X to the base 10, and has the same data type as X. X must be a floating-point value greater than zero.

▶ LOG2(X)

The result is the logarithm of X to the base 2, and has the same data type as X. X must be a floating-point value greater than zero.

▶ MAX(X,Y)

X and Y must be arithmetic values.

The values of X and Y are converted to a common arithmetic type using the conversion rules for arithmetic infix operators. The result is the larger of these converted values and has the common arithmetic type.

▶ MIN(X,Y)

X and Y must be arithmetic values. The values of X and Y are converted to a common arithmetic type using the conversion rules for arithmetic infix operators. The result is the smaller of these converted values and has the common arithmetic type.

▶ MOD(X,Y)

X and Y must be arithmetic values.

The result is the truncated remainder of X divided by Y. The result has the common type of X and Y.

Let (px,qx) and (py,qy) represent the precision of X and Y respectively. If the common type of X and Y is fixed-point, the precision of the result is:

(MIN(N,py-qy+MAX(qx,qy)),MAX(qx,qy)).

Otherwise, the precision of the result is MAX(px,py). N is the maximum precision allowed for the common type of X and Y.

If Y = Ø, the result is X; otherwise, the result is X-Y*FLOOR(X/Y).

For example, the result of MOD(15,2) is 1.

▶ NULL() or NULL

The result is a null pointer value.

▶ ONCODE() or ONCODE

The value returned by this function is a fixed binary integer that indicates the reason why the condition was signalled. (For example, an error code.) Its value is 0 if no on-unit is currently active.

The values returned and their precision are implementation-defined.

▶ ONFILE() or ONFILE

The value returned by this function is a character string containing the filename for which the most recent ENDFILE, KEY, or ENDPAGE condition was signalled. Its value is a null string if no on-unit for one of those three conditions is currently active.

▶ ONKEY() or ONKEY

The value returned by this function is a character string containing the KEY value for which the most recent KEY condition was signalled. Its value is a null string if no on-unit for the KEY condition is currently active.

▶ ONLOC() or ONLOC

The value returned by this function is a character string containing the name of the procedure which was executing when the most recent condition was signalled. Its value is a null string if no on-unit is currently active.

▶ PAGENO(X)

The result is a fixed binary integer giving the current page number in the file control block identified by X.

The precision of the result is implementation-defined.

If X does not identify an open file control block with the PRINT attribute, the program is in error.

▶ RANK(X)

A PL1G extension. X must be a character-string of length one.

The result is a fixed binary integer giving the position of the character within the collating sequence. The result is defined as RANK(X) = INDEX(COLLATE,X)-1, and has an implementation-defined precision. The resulting value is formed by taking the bits of X as an unsigned binary integer value. On computers that use the ASCII character set with the high order bit of each byte set to '1', the definition of this function is INDEX(COLLATE(),X)+127.

▶ ROUND(X,K)

X is the arithmetic value to be rounded, K is a signed integer constant indicating the position within X to be rounded.

The result is the value of X rounded such that the K-th position of X is expressed to its nearest integer.

For example, the result 3.215 is returned for ROUND(3.2146,3).

▶ SIGN(X)

The result is a fixed binary integer -1, 0, 1 indicating the sign of X. The precision of the result is implementation-defined.

▶ SIN(X)

The result is the sine of the angle X expressed in radians, and has the same data type as X. X must be a floating-point value.

▶ SIND(X)

The result is the sine of of the angle X expressed in degrees, and has the same data type as X. X must be a floating-point value.

▶ SINH(X)

The result is the hyperbolic sine of the angle X expressed in radians, and has the same data type as X. X must be a floating-point value.

▶ SQRT(X)

The result is the positive square root of X, and has the same data type as X. X must be a positive nonzero floating-point value.

▶ STRING(S)

S is an arithmetic or string value or an array or structure containing all string values and suitable for storage sharing as defined in Section 4. S must be a reference to a variable whose storage is connected.

The result is S converted to a string according to the rules for type conversion given in Section 8.

▶ SUBSTR(S,I,J) or SUBSTR(S,I)

S is either a bit or character-string, I must be a fixed-point integer value indicating the first bit or character of a substring within X, J must be a fixed-point integer value indicating the length of the substring. If J is not given, then J = LENGTH(S)-I+1.

The result is a string that is a copy of a part of the string s starting at the I-th character for a length J.

The program is in error if I<1 or (I+J-1)> LENGTH(S) or J<Ø. If the program is compiled with subscript checking enabled, these errors result in the ERROR condition; otherwise, these errors produce unpredictable results.

▶ TAN(X)

The result is the tangent of the angle X expressed in radians, and has the same data type as X. X must be a floating-point value.

▶ TAND(X)

The result is the tangent of the angle X expressed in degrees, and has the same data type as X. X must be a floating-point value.

▶ TANH(X)

The result is the hyperbolic tangent of the angle X expressed in radians, and has the same data type as X. X must be a floating-point value.

▶ TIME() or TIME

The result is a character string of an implementation-defined length of at least six characters of the form HHMMSS[FFF...] that represents the time of day, where HH, MM, and SS are in the ranges ØØ:23, ØØ:59, and ØØ:59, and represent hours, minutes, and seconds respectively. If an implementation returns a string of length greater than six, FFF... represents decimal fractions of a second.

▶ TRANSLATE(S,T) or TRANSLATE(S,T,X)

S is a character-string, T is a character-string, and X is also a character-string. If T is shorter than X, T is padded on the right with blanks until the length of T is equal to the length of X.

If X is not given, it is assumed to be COLLATE().

The occurrence of an element of X in the string S is replaced by the corresponding element in the string T.

If S is the null string, the result is the null string. If S is not the null string, then for each character of S, S(k), the value i is calculated to be equal to INDEX(X,S(k)). If the value of i is Ø, the corresponding character of the result is S(k); otherwise, the corresponding character of the result is T(i).

For example, the result of TRANSLATE('1#2#','Ø','#') is '1Ø2Ø'.

▶ TRUNC(X)

The result is the integer part of X.

If X is a floating point value, the precision of the result is the precision of X; otherwise, the precision of the result is (MIN(N,MAX(p-q+1,1)),Ø). N is the maximum precision allowed for fixed-point values of the result type.

For X<Ø, the result is CEIL(X). For X>=Ø, the result is FLOOR(X). Examples:

The result of TRUNC(3.125) is 3.

The result of TRUNC(-3.125) is -3.

▶ UNSPEC(X)

X must be a reference to a scalar variable.

The result is a bit-string containing the internal representation of X. The result value's length and content depend on the type and value of X and are implementation-defined.

▶ VALID(X)

X must be a reference to a scalar pictured value.

The result is a bit-string of length one that indicates if the character-string value of X can be edited into the picture declared for X.

The result value is '1' B if the character-string value of X can be edited into the picture declared for X; otherwise, the result is 'Ø'B.

▶ VERIFY(S,C)

S and C must be character-string values.

The result is Ø if each of the characters in S occurs in C. Otherwise, VERIFY returns an integer that indicates the leftmost character in S which is not found in C.

For example, verify('2a56b','Ø123456789') returns the value 2 to indicate the first non-numeric character in the string '2a56b'.

Part III
PL1G and the Prime System

SECTION 11

IMPLEMENTATION DEFINED FEATURES

This section defines those PL/I Subset G properties that are dependent on a particular implementation. Programs whose correct execution depends on these properties may require modification in order to execute correctly on another implementation.

ARITHMETIC PRECISION

Each arithmetic data type has a default precision and a maximum precision as specified by the following table.

Data Type	Maximum Precision	Default Precision
Fixed Binary	31	15
Fixed Decimal	14	5
Float Binary	47	23
Float Decimal	14	6

The range of the scale factor of fixed-point decimal data is $0 \le q \le 14$.

MAXIMUM SIZES

- The maximum length of a string value is 32767 bits or characters. However, any string expression that produces an intermediate result requires storage that is allocated either on the stack or in the system storage area used to support the ALLOCATE statement. Allocation of large temporary storage blocks exceeding the amount of available storage results in a signal of the ERROR condition.

- The maximum size of a string constant is 256 bits or characters. This limit is checked by the compiler after expansion of bit string constants that are written in B2, B3, or B4 format.

- The maximum size of an internal array is 64K words. However, external static arrays may be up to 2**31 words long, subject to available memory constraints.

- The default value of the LINESIZE option of an OPEN statement is 120.

- The default value of the PAGESIZE option of an OPEN statement is 60.

- The COLLATE built-in function returns a string consisting of the 128 characters in the ASCII character set.

- The maximum length of a name is 32 characters. However, external names longer than eight characters are truncated to that length and a warning message is printed at compile time.

- The maximum length of a string value transmitted by a GET or PUT statement is 256 bits or characters.

DATA SIZE AND ALIGNMENT

The size and alignment of each PL1G data type is given by the following table:

Data Type	Alignment	Size (p = precision)
Fixed Binary(p)	word	1 word p <= 15
Fixed Binary(p)	word	2 words p > 15
Fixed Decimal(p)	byte	(p+2)/2 bytes
Float Binary(p)	word	2 words p <= 23
Float Binary(p)	word	4 words p > 23
Float Decimal(p)	word	2 words p <= 6
Float Decimal(p)	word	4 words p > 6
Character(n)	byte	n bytes
Character(n) Varying	word	(n+3)/2 words
Bit(n)	bit	n bits
Bit(n) Aligned	word	(n+15)/16 words
Pointer	word	3 words
Pointer Options (short)	word	2 words
Picture	byte	n bytes
Label	word	4 words
Entry	word	4 words
File Constant	word	55 words
File Variable	word	2 words
Structure	max of members	sum of members

Data not contained in a structure or array is allocated on a word or double word boundary.

The ALIGNED attribute applied to character data has no effect on the alignment of the data.

INPUT/OUTPUT ON TTY

The device named TTY is unlike any other stream file in the following respects:

- Each PUT statement transmits data to the device without waiting for the line to be filled.

- A GET statement resets the current column position used by subsequent PUT statements.

READ AND WRITE ON STREAM FILES

READ and WRITE statements can operate on stream files if they specify a scalar varying character string variable in their INTO and FROM options. A READ statement reads the next complete input line and assigns it to the varying character string specified by the INTO option. The string does not include any new-line character. A WRITE statement puts any partial line currently in the output buffer of the file and then writes a line consisting of the current value of the varying string specified by the FROM option.

The use of READ and WRITE statements on stream files is an easy and efficient method of processing variable length lines. However, this feature is not part of standard PL/I and makes your program dependent on those implementations that support this feature.

VARIABLE LENGTH INPUT LINES

A stream input file consists of a sequence of lines each of which may have a unique length. Lines read from a disk file are not modified in any way. Lines read from all other devices have trailing blanks removed and have one blank appended to the end of each line that is read by a GET statement. This blank ensures that a field typed at the end of a line is not appended to a field typed at the beginning of the next line.

A stream file containing variable length input lines cannot be processed by standard PL/I. Two nonstandard methods can be used to read these files. A READ statement can be used to read lines as described by the previous paragraphs. An edit-directed GET statement can also be used. In order to read a variable length line with a GET statement, an A-format without a field width must be used. The A-format used in this way reads the content of the current line beginning with the current column and ending with the end of the line as its input field. It then sets the column position so that the next operation will read a new line.

THE TITLE OPTION AND FILE OPENING

If a file opening is performed without a TITLE option, the file description, fd, is obtained by taking the file-id from the file control block. The file-id of a file control block is the name of the file constant associated with that file control block.

If a file opening is performed with a TITLE option, the file description, fd, is the character string value given by the TITLE option.

Regardless of how the fd is obtained, it must have one of the following general forms:

```
name
name -SAM [n]
name -DAM [n]
name -APPEND [n]
name -DEVICE [n]
```

where n is an integer that gives the maximum record size in words. One or more blanks must separate the name from the rest of the file description. Blanks must also separate the record size from the file or device type. The record size is ignored for stream files and need not be given. If omitted for a record file, a default of 1Ø24 words is supplied.

-SAM n specifies that name is the name of a disk file that contains variable length records written without keys. The maximum length of any record in the file is n words.

-DAM n specifies that name is the name of a disk file that contains variable length records written with integer keys. The maximum length of any record in the file is n words. Each record is stored in n words to allow for quick access of the record using its key. These files cannot be read as sequential files, they must be opened using the DIRECT file attribute.

-APPEND n has the same meaning as -SAM, except that it causes the output produced by this opening to be appended to the end of any existing SAM file. -SAM n causes any existing output file to be deleted and a new file created.

-DEVICE n specifies that name is the name of a device whose maximum record size is n words. Only nonkeyed files can be read or written to devices. The following table gives the possible device names and the type of file that can be read or written to the device.

Device Name	File Type	Input/Output
SYSIN	Stream	Input from TTY
SYSPRINT	Stream	Output to TTY
TTY	Stream	Input/Output to TTY
PTR	Stream	Input Paper Tape
PTP	Stream	Output Paper Tape
CR	Stream	Input Card Reader
SPR	Stream	Output Serial Printer
MTØ	Stream/Record	Input/Output Magnetic Tape
MT1	Stream/Record	Input/Output Magnetic Tape
MT2	Stream/Record	Input/Output Magnetic Tape
.	.	.
.	.	.
.	.	.
MT7	Stream/Record	Input/Output Magnetic Tape
PRØ	Stream	Output Line Printer Ø
PR1	Stream	Output Line Printer 1

If a specification of -DAM, -SAM, -DEVICE, or -APPEND is not given, the following rules are used to make the opening:

- If the name is a device name, -DEVICE is used.
- If the opening is for a DIRECT file, -DAM is used.
- If neither of the previous cases was true, -SAM is used.

The effect of these rules is that a specification need only be given if the maximum record size exceeds 1024 words, if a device name is to be used as the name of a disk file, or if an output file is to be appended to the end of an existing file.

A DIRECT is always a -DAM file and must not be opened as a -SAM, -APPEND, or -DEVICE file.

LISTING CONTROL

The %NOLIST; statement causes the compiler to stop placing listing information into the listing file, if any; %LIST; causes the compiler to resume its former operation with respect to the listing. These directives can be used with %INCLUDE files, for example, to exclude a long repetitive included declaration from the generated listing. Another possible use is to suppress uninteresting portions of an expanded (i.e., including the generated code) listing.

POINTER SIZE CONTROL

The default size for pointers is three words, allowing unaligned character- and bit-strings to be addressed. However, some applications may desire a two-word pointer (which cannot address unaligned strings), either because of the space-critical nature of the application or because of an interface with existing software. These applications may declare pointers as POINTER OPTIONS(SHORT). However, use of these pointers either to address unaligned data or to receive the address of unaligned data is a program error that is not diagnosed by the compiler, which will cause anomalous and unpredictable program behavior.

ADDITIONAL IMPLEMENTATION DEFINED FEATURES

- Tab stops are set every 7 columns of a stream file that has the PRINT attribute.
- Two exponent digits are produced when a floating-point value is converted to a character string value.
- The file-name used in a %INCLUDE must be a valid PRIMOS file name enclosed in quotes.

- Keys are restricted to 32 characters. The variable used in a KEYTO option must be a varying character string.

- All built-in functions that are described in Section 10 as producing a fixed-point binary integer result of implementation defined precision produce a result whose precision is 15.

- The character set of the computer is ASCII stored with the high order bit of each byte set to one. This is the collating sequence used for comparisons of character data. Because the high order bit is set to one, the definition of the RANK and BYTE built-in functions differs from that used in several other implementations of PL/I that also support RANK and BYTE.

NULL BUILT-IN FUNCTION

The value of the NULL() built-in function is 7777(0)/0. The 7777 represents a fictitious segment whose number is higher than that of any actual segment. The (0) is a ring number, and the final 0 is a word number.

FILE SYSTEM LIMITATION

PRIMOS allows the user to access up to 128 files. In PL1G, the user may access at most 16 files concurrently.

SECTION 12

ADVICE ON THE USE OF PL1G

PURPOSE OF THIS SECTION

This section is intended for experienced as well as new users of PL/I. It provides advice on how to write readable and efficient PL1G programs, and information which may be useful when programs fail for no obvious reason.

EFFICIENCY

Input/Output

Record I/O is nearly always faster and requires less space than stream I/O. Record I/O transmissions frequently can be performed without copying the record into a PL1G I/O system buffer.

Record I/O using integer keys is often more efficient than using character-string keys.

List-directed I/O requires less execution-time support code and generally is faster than edit-directed I/O, but the difference may not be significant.

Arithmetic

Because computers use binary integers for addressing data, fixed-point binary is the most efficient form of arithmetic. Floating-point arithmetic with either binary or decimal precision is also generally supported by hardware and is efficient. Fixed-point decimal data may be implemented in many ways, but is normally stored as packed decimal data. Operations on decimal data nearly always require more time and more space, frequently by a factor of 5 when compared with fixed binary arithmetic.

Variable Size Data

Constant size data can be more easily addressed and operated upon than variable size data. Whenever possible, declare the size of strings and the bounds of arrays as integer constants.

If a structure contains one or more members with a variable size, place all such members at the end of the structure following all members with fixed sizes. This improves the addressing of the fixed size members.

If possible, make the sizes of based variables constant. If they must be variable, make them simple unsubscripted variable references rather

than expressions. The size of a based variable is evaluated for each reference to the based variable.

COMMON PROGRAMMING ERRORS

The compiler is capable of producing approximately 225 unique error messages. However, some errors are too difficult or too costly to diagnose either during compilation or during execution of a program. Other errors may produce misleading diagnostics. The following is a list of the most common errors that are either not diagnosed or that produce misleading error messages.

- Comments that do not end with */ cause the compiler to consider all text up to the next */ as part of the comment. The compiler prints an * next to the line number of each line that is a continuation of a comment. Examine your listing for these asterisks and ensure that all comments are properly closed.

- String constants that do not end with a quote cause similar problems as unclosed comments, but * is not used to indicate continued strings. Because line boundaries do not necessarily consist of a carriage return or a new-line character, character string constants should not be continued across a line boundary. In some implementations, the carriage return, new-line, or both are considered part of the program text, but on other systems they are not.

- Statements that do not end with a semi-colon cause incorrect recognition of the statement type. This normally produces a reasonable error message.

- %REPLACE statements that replace keywords cause incorrect recognition of the statement or option designated by that keyword. Because the listing looks correct, these errors are hard to understand.

- Declarations of procedures in other program modules that disagree with the actual PROCEDURE statements in those modules. Since the loader generally does not check for mismatches, such errors may go undetected.

- Pointers used with based variables that are not valid descriptions of the object designated by the pointer. These errors cannot be detected by the compiler and are too costly to detect during program execution. See Section 4 which describes based variables and storage sharing.

- Null pointer values used to access based variables produce an ERROR condition in some implementations, but are not detected in others.

- Subscripts or arguments of SUBSTR that are out-of-range are detected only if checking has been requested by use of a

compile-time option. When detected, they cause the ERROR condition to be signalled.

- Using the value of a variable without first assigning a value to the variable is not detected and may produce unpredictable results some of which appear to be "correct".

- Calculating values that exceed the precision of the variable into which they are stored is an error that may cause the ERROR condition, but that is not detected by most implementations.

- Calling a procedure recursively without giving it the RECURSIVE option may produce unpredictable results.

PROGRAMMING STYLE

Programs that are hard to read generally contain many more errors than programs that are easy to read. By following a few simple rules a programmer can ensure that his programs can be read and maintained by others, and that he can understand his own code months or years after it was written.

The recommendations given in this section represent the author's programming style and is the style used by the examples in this manual. Minor variations on this style are widely used by software specialists who write in PL/I.

Procedures

Procedures are a natural division of the program and provide the best mechanism to represent program modularity. The problem to be programmed should be organized or structured as a set of modules each of which is written as a procedure.

Procedures should be written with a minimal dependence on their ability to access the variables of their containing procedure. All such use of nonlocal variables should be clearly documented by comments in both the declaring and using procedures.

Procedures used only from within a given procedure should be written as an internal procedure within the calling procedure. Internal procedures should be written following the executable statements of the containing procedure.

In order to facilitate understanding of its logic, each procedure should be as small as possible. A procedure, excluding any contained procedures and declarations, should seldom exceed two pages of text or approximately 150 lines. Extensive use of internal procedures is recommended and may result in external procedures of thousands of lines. Because all compilers have some limits on the size programs that they can compile, it is wise to keep external procedures shorter than 10,000 source lines.

Formatting Rules

The most important aspect of programming style is the way in which the text of a program is formatted. A consistently formatted program can be easily read by its author and other programmers who may need to work with the program. A badly formatted program is nearly unreadable. The rules given here are easy to follow and have been successfully used by many programmers.

1. Place the label prefix of an executable statement on the line immediately before the body of the statement and begin the label in column one. This makes labels easy to find and allows insertion of additional statements after the label.

2. Begin the body of each executable statement one tab stop from the left margin. This makes the labels easy to find and clearly separates the executable statements from the declarations.

3. Place all DECLARE statements at the head of their containing procedure and begin them in column one. Use the keyword DECLARE rather than DCL.

4. Indent the keywords THEN and ELSE and align them with each other. Also indent the statements contained in a THEN or ELSE clause so that any END statement that closes a clause is aligned with the DO that heads the clause. Example:

```
IF A < B
   THEN DO;
        .
        .
        END;
   ELSE IF P = NULL
           THEN STOP;
           ELSE DO;
                .
                .
                .
                END;
```

5. An IF statement used to select one of many cases should have its ELSE clause aligned with the IF. Example:

```
IF A = 1
   THEN DO;
        .
        .
        .
        END;
ELSE IF A = 2
   THEN DO;
        .
        .
        .
        END;
ELSE IF A = 3
   THEN DO;
        .
        .
        .
        END;
ELSE ...
```

6. Do not write more than one statement on a line.

7. Indent the body of a do-group so that its DO and END statements are aligned with each other and so that the statements contained within the group are indented further than the DO and END statements. This allows the reader to easily find the matching END statement for any DO. However, as shown in the previous examples, a do-group that serves as a THEN or ELSE clause is already clearly indented and requires no further indentation.

8. END statements that end a procedure should contain the procedure name either as a comment or as a closure label. However, if your compiler inserts additional END statements without giving any warning, do not use the closure label.

9. Use blank lines to separate blocks of comments from blocks of program text so that the comments can be easily identified.

1Ø. Use redundant parentheses to establish the order of evaluation of expressions that contain both & and | operators. They should also be used in any expression containing operators of more than two or three different levels of priority. Such expressions are difficult to read because most programmers cannot remember the exact priority of all operators.

11. Also use redundant parentheses in the following two cases:

```
X = (Y = Z);
R = -(P->S.M);
```

12. Use only the forms of the DECLARE statement that are recommended in Section 5.

Names and Abbreviations

Use meaningful names of up to 32 letters for all important variables and for all procedures. For very local variables that are used as do control variables, subscripts, counters, etc., use short meaningless names such as I, J, K, etc.

Programs are easier to read if they contain full keywords rather than abbreviations. If abbreviations are used for long keywords such as CHARACTER, use them consistently.

When referencing members of a structure, use the major structure name and the member name. If subscripts are used in a structure qualified reference, place each subscript after the name that has the corresponding dimension.

Use an explicit pointer qualifier when referencing a based variable, unless all references to that variable are based on the same pointer. In the later case, declare the variable to be based on the pointer and use implicit pointer qualification as explained under BASED STORAGE in Section 4 and POINTER QUALIFIED REFERENCES in Section 6.

SECTION 13

PL1G USE OF THE CONDITION MECHANISM

The use of the PRIMOS condition mechanism is discussed in detail in The Prime Users Guide. The procedures that may be called to implement features of the condition mechanism are described in The PRIMOS Subroutines Reference Guide. This section discusses details specific to PL1G's use of the condition mechanism.

INFORMATION STRUCTURE

The standard PL/I information structure is used to communicate the value of the ONCODE builtin function. This structure is as follows:

```
DCL 1 INFO BASED,
        2 FILE_PTR POINTER,
        2 INFO_STRUCT_LENGTH FIXED BIN,
        2 ONCODE_VALUE FIXED BIN,
        2 RET_ADDR POINTER;
```

When invoked, the condition mechanism searches the stack backwards until it finds a condition frame. This condition frame is used to determine the value to be returned. If no condition frame is found, the value 0 is returned.

The value of the ONLOC built-in function (see Section 10) is determined by searching the stack backwards until a condition frame has been found. The backward search continues, looking for the first PL1G procedure frame whose name does not begin with F$, P$, or I$, since these characters are used to denote run-time support (library) routines. (A PL1G procedure frame is known by Bit 5 of the FLAGS word of the stack frame being set. The procedure name is indicated by the owner pointer of all PL1G stack frames, since it points to the ECB of the procedure, which is followed by the name of the procedure). The name of that procedure is returned as the value of the function.

The values of the ONFILE and ONKEY built-in functions (see Section 10) are determined by finding the information structure described in in the previous paragraphs and using the file pointer in the information structure to extract the data from the file control block.

The actual values of the ONCODE built-in function are subject to change. They are intended for informational purposes only, and users are strongly advised <u>not</u> to make program control decisions based on these values. However, the file named SYSCOM>ONCODES.PL1 is provided for those users who wish to allow an on-unit to print the additional explanatory information normally given by the system default on-unit.

The following program fragment will accomplish that function:

```
/* Previous program text */

ON ERROR BEGIN;

    %INCLUDE 'SYSCOM>ONCODES.PL1';
    DECLARE ONCODE_VALUE FIXED BIN;

    ONCODE_VALUE = ONCODE();

    IF ONCODE_VALUE > Ø & ONCODE_VALUE <= MAX_IO_ONCODE
        THEN PUT SKIP LIST
        ('ERROR SIGNALED: ' || IO_ONCODE_MESSAGE(ONCODE_VALUE));

    ELSE IF ONCODE_VALUE >= ONCODE_BASE & ONCODE_VALUE < ONCODE_BASE +
    NEXT_AVAILABLE_CODE
        THEN PUT SKIP LIST
        ('ERROR SIGNALLED: ' || ONCODE_MESSAGE(ONCODE_VALUE -
        ONCODE_BASE + 1));

    ELSE PUT SKIP LIST
    ('ERROR SIGNALED: (NO AVAILABLE ONCODE() VALUE)');

    PUT SKIP;
    GO TO RECOVERY_LABEL;

    END;
```

SECTION 14

USING THE PL1G COMPILER

INTRODUCTION

Prime's compiler accepts a source program meeting the PL/I Subset G or the PL1G standard. It can output a source listing, error and statistics information, an object file, and various messages. Errors are printed at the terminal as the compiler detects them.

This section tells:

- How to invoke the compiler
- How to specify options to the compiler
- The significances of the various messages that are printed during compilation
- The meanings of the various compiler options

INVOKING THE COMPILER

The PL1G Compiler is invoked by the PL1G command to PRIMOS:

PL1G pathname [-option 1] [-option 2] . . . [-option n]

pathname The pathname of the PL1G source program to be compiled.

options Mnemonics for the options controlling compiler functions.

All mnemonic options begin with a dash "-". Example:

PL1G TEST1 -RANGE -DEBUG -LISTING

will cause TEST1 to be compiled with the options given.

COMPILER ERROR MESSAGES

For each error encountered in the program, an error message will be printed at the terminal and in the source listing if one exists. The general format of an error message is:

ERROR xxx SEVERITY y BEGINNING ON LINE zzz
explanation

xxx	Error Code
y	Severity code
zzz	Line number where error begins
explanation	Description of the error, and possible remedies.

The significance of the severity code is:

Severity	Description
1	Warning.
2	Error that has been corrected.
3	Uncorrected error - prevents optimization and code generation.
4	Error that prevents further compilation.

PL1G Error Messages are self-explanatory. They are not listed in this guide, since such a listing could only repeat information already given in the individual messages.

END-OF-COMPILATION MESSAGE

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

xxxx ERRORS (PL1G-REV ZZ.Z)

xxxx	The number of compilation errors (0000 indicates a successful compilation)
ZZ.Z	The current revision number of the PL1G compiler

After compilation, control returns to the PRIMOS level.

COMPILER OPTIONS

The available compiler options can be categorized as follows:

- Specify the source file
- Specify the existence and contents of the source listing
- Specify the handling of error and statistics information
- Specify the existence and properties of the object code

Compiler options generally come in pairs: for each one, there is a converse option having the opposite effect. Most option pairs direct the compiler to do/not-do some action. A few present a choice between two actions. One member of each pair is always the default.

Not all options can be specified explicitly. When either member of an option pair could be a desirable default at some installation, both options are explicitly available, so that the default can always be reversed. When only one member could be a desirable default, that option cannot be explicitly specified; it is selected by simply accepting the default.

In the following list, each option is given along its converse. Options which cannot be given explicitly are printed in lowercase without an initial dash. For each pair, the Prime-supplied default is underlined. Commonly used options are marked with an asterisk: new users should skip over the unasterisked options.

Some options require an argument in addition to the option specification. The argument follows the option, and is <u>not</u> preceded by a dash. Options may be given in any order.

Table 14-1 lists the options in the order that they are discussed below. At the end of this section, Table 14-2 lists them alphabetically with their abbreviations, to provide a quick reference.

Table 14-1. Compiler Options

Specify the Source File

	-S and -I	Give name of source file
	-UPCASE / -LCASE	Convert source file to upper case

Specify the Existence and Contents of the Source Listing

*	-L [argument]	Controls existence of listing file
*	-XREF / noxref	Cross reference in source listing
	-EXPLIST / noexplist	Assembly code in source listing
	-OFFSET / nooffset	Offset map in source listing
	-NESTING / nonesting	Nesting level in source listing

Specify the Handling of Error and Statistics Information

	-SILENT / nosilent	Suppress Warning Messages
	-STATISTICS / nostatistics	Print compilation statistics

Specify the Existence and Properties of the Object Code

*	-B [argument]	Controls existence of object file
	-BIG / nobig	Controls dummy array handling
	-64V / -32I	Controls addressing mode
*	-DEBUG / nodebug	Controls generation of debugger code
*	-OPTIMIZE / -NOOPTIMIZE	Controls optimization
	-PRODUCTION / noproduction	Controls generation of debugger code
*	-RANGE / norange	Inserts range-checking code

* Indicates options most useful to new users.

Prime-supplied defaults are underlined.

Specify the Source File

The source file is usually designated by pathname immediately after the PL1G command. Alternatively, it may be given in an option. Lowercase letters in the source can be automatically mapped to uppercase before compilation.

▶ -S[OURCE] pathname and -I[NPUT] pathname

Either of these can be used to designate the source file to be compiled, as an alternative to naming the file immediately after the PL1G command. The following are equivalent:

```
PL1G pathname -RANGE -BIG

PL1G -RANGE -BIG -I pathname

PL1G -BIG -S pathname -RANGE
```

The pathname must not be designated more than once.

▶ -UPCASE / -LCASE

Controls mapping of lowercase to uppercase letters in a source program.

-UPCASE: Any lowercase letters in the source will be treated as uppercase by the compiler, except in character constants.

-LCASE: Lower and uppercase letters remain distinct.

Specify the Existence and Contents of the Source Listing

The PL1G compiler's primary output to the programmer is the source listing. When the -L option is given, a basic source listing is created, containing:

- Date and time of compilation
- Options in effect
- Source text
- External entry points
- Symbol-Table Listing
- List of errors

Additional options can be given, to cause additional data to be inserted into the source listing: a cross reference, offset map, or pseudo-assembly code listing may be included. If such an option is given but no source listing was specified, -L YES will be assumed.

▶ * -L[ISTING] [argument]

Controls creation of the source listing file. The argument may be:

pathname	Listing will be written to the file pathname.
YES	Listing will be written to a file named L_program, where program is the name of the source file.
TTY	The listing will be printed at the user terminal.
SPOOL	The listing will be spooled directly to the line printer. Default SPOOL arguments are in effect.
NO	No listing file will be generated.

When no -L option is given, -L NO will be presumed. When -L is given with no argument, -L YES will be presumed.

▶ * -XREF / noxref (Implies -L)

Controls generation of a cross reference

-XREF: A cross reference will be appended to the source listing. A cross reference lists, for every variable, the number of every line on which the variable was referenced.

noxref: No cross reference will be generated.

▶ -EXPLIST / noexplist (Implies -L)

Inserts a pseudo-assembly code listing into the source listing.

-EXPLIST: Each statement in the source will be 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, FDR3059.

noexplist: No assembler statements are printed.

▶ -OFFSET / nooffset (Implies -L)

Appends an offset map to the source listing.

-OFFSET: An offset map is appended 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.

nooffset: No offset map is created.

▶ -NESTING / nonesting (Implies -L)

Includes logical control nesting level in the source listing.

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

nonesting: No nesting numbers are produced.

Specify the Handling of Error and Statistics Information

Level 1 error messages (warnings) can be suppressed if desired. Compiler statistics can be printed at the terminal after each phase of compilation, but not to a user file other than a COMOUTPUT file.

▶ -SILENT / nosilent

Suppresses WARNING messages.

-SILENT: Level 1 Error Messages will not be printed at the terminal, and will be omitted from any listing file.

nosilent: Level 1 Error Messages are retained.

▶ -STATISTICS / nostatistics

Controls printout of compiler statistics.

-STATISTICS: A list of compilation statistics is printed 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.
- PAGING Disk I/O time.
- CPU CPU time in seconds, followed by the clock time when the phase was completed.

nostatistics: Statistics are not printed.

Specify the Existence and Properties of the Object Code

For a given source program, the compiler can produce a variety of object programs or none at all, depending on the options given. The areas open to programmer control are:

- Creation of the object file
- Storage allocation and addressing
- Compiler augmentation of the object code

Creation of the Object File: The -B option controls the existence and naming of the object file, but not the properties it will have.

▶ * -B[INARY] [argument]

The argument may be:

pathname	Object code will be written to the file pathname.
YES	Object code will be written to the file named B_program, where program is the name of the source file.
NO	No binary file will be created. Specified when only a syntax check or source listing is desired.

When no -B option is given, or -B without an argument is given, -B YES will be presumed.

Storage Allocation and Addressing: By giving appropriate options, the programmer can cause compiled subprograms to handle BASED and PARAMETER aggregates longer than a segment, and can determine the addressing mode (64V or 32I) to be used in the object file.

▶ -BIG / nobig

Determines code generated for BASED or PARAMETER aggregate references in a subprogram.

-BIG: A BASED or PARAMETER aggregate can become associated with any aggregate.

nobig: A BASED or PARAMETER aggregate can become associated only with an aggregate that does not cross a segment boundary.

▶ -64V / -32I

These determine the addressing mode to be used in the object code. 64V is a segmented virtual addressing mode for 16-bit machines. 32I is a segmented virtual mode which takes maximum advantage of the 32-bit architecture of Prime's more advanced models (P450 and up). R and S modes (relative and sectored addressing) are not available for PL1G.

Augmented Object Code

When no augmented-code options are given, the source program is compiled statement by statement, and the resulting object code becomes the object file. Alternatively, the compiler can optimize the object code, and can add additional code to provide range checking or the capacity to run under the symbolic debugger.

▶ * -DEBUG / nodebug

Controls generation of code for the debugger.

-DEBUG: The object file is modified so that it will run under the symbolic debugger. Execution time is increased. The code generated will not be as highly optimized.

nodebug: No debugger code is generated.

▶ * -OPTIMIZE / -NOOPTIMIZE

Controls the optimization phase of the compiler.

-OPTIMIZE: The object code will be optimized. Optimized code runs more efficiently than non-optimized code, but takes somewhat longer to compile.

-NOOPTIMIZE: Optimization does not occur.

▶ -PRODUCTION / noproduction

Alternative option controlling code for the debugger.

-PRODUCTION: Similar to DEBUG, except that the code generated will not permit insertion of statement break points. Execution time is not affected.

noproduction: Production-type code is not generated.

▶ * -RANGE / norange

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

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

norange: Out-of-bounds values will not be detected. The program will be more vulnerable to errors, but will execute more quickly.

OPTION ABBREVIATIONS

The PL1G compiler options may be abbreviated, as follows:

The abbreviations -L, -B, -I, and -S stand for -LIST, -BINARY, -INPUT, and -SOURCE, respectively, regardless of what other abbreviations are used.

Except where the above rule takes precedence, the abbreviation for any compiler option is the shortest string of leftmost characters from the option's name that uniquely identify the option. Any number of additional characters, up to the complete name, may also be given.

These rules produce the abbreviations shown in Table 14-2. The table is also intended to provide a quick alphabetical reference for those already familiar with the compiler options.

Table 14-2. Summary of Compiler Options and Abbreviations.
(Defaults are underlined.)

Option	Abbreviation	Significance
-BIG	-BIG	Boundary-spanning code
-BINARY	-B	Creation of object file
-DEBUG	-DE	Debugger code
-EXPLIST	-EX	Expanded source listing
-INPUT	-I	Designate source file
-LCASE	-LC	No source-file case conversion
-LIST	-L	Creation of source listing
-NESTING	-NE	Indicate nesting level
-NOOPTIMIZE	-NOOP	Don't optimize object code
-OFFSET	-OF	Offsets in source listing
-OPTIMIZE	-OP	Optimize object code
-PRODUCTION	-P	Generate production code
-RANGE	-R	Check subscript ranges
-SILENT	-SI	Suppress warning messages
-SOURCE	-S	Designate source file
-STATISTICS	-ST	Print compiler statistics
-UPCASE	-U	Convert to uppercase
-XREF	-X	Generate cross-reference
-32I	-3	Produce 32I mode code
-64V	-6	Produce 64V mode code

Appendices

APPENDIX A

GLOSSARY OF PL1G TERMS

- ALIGNED

An attribute specifying that the declared bit string data is to be aligned on a convenient storage boundary, and, if necessary, to use more bits of storage than are specified by the bit string's declared length.

- ALLOCATE

A statement causing storage to be set up for the based variable named in the statement and setting a pointer to the storage allocated.

SET Clause: A clause of the ALLOCATE statement specifying a pointer variable whose value is to be set and which identifies the generation of storage for the based variable.

- Argument Passing

Arguments are passed by reference and by value.

By Reference: Where the parameter and the argument describe the same storage, the argument is said to have been passed by reference.

By Value: If the argument is an expression, function reference, built-in function reference, constant, parenthesized variable reference, or a reference to a variable with a data type that does not match the parameter, then, the argument is copied to a temporary block of storage in the caller's stack frame, which is then passed to the parameter instead of the actual argument; the argument is said to be passed by value.

- Arithmetic Data

A value whose data type has base, scale, and precision, and can be used in computation.

- Array

An n-dimensional ordered set of values (elements) all having the same data type.

Elements of an array are referenced by their position within an array. Each array has a specified (declared) number of dimensions and each dimension has a specified lower and upper bound.

- Array Bound

That component of the dimension attribute that defines the upper or lower limit of a dimension of an array.

- Array Dimensioning

The subdivision of array elements into logical sets, i.e., each dimension of an array specifies the number of elements in a logical set.

- Arrays of Structures

Structures, like other variables, can be declared as arrays and may contain arrays as members.

- Assignment

A statement used to evaluate expressions on the right of the assignment symbol (=) and assign the result of the evaluation to variables or pseudo-variables on the left of the assignment symbol (i.e., the target of the assignment).

- Attribute

A property that can be associated with an identifier that characterizes the identifier or the object represented by the identifier.

- Attribute Factoring

Enclosure of a list of identifiers and partial attribute sets in parentheses followed by the set of attributes which are common to all of them; i.e., repeated specification of the same attribute for many identifiers.

- AUTOMATIC

An attribute specifying allocation of storage for the associated variable on each entry to the block which contains the declaration and release of storage upon exit from that block.

- Base

Either BINARY or DECIMAL.

- BASED

An attribute specifying that the address of the storage referenced is supplied by a pointer value. This storage may be managed by the ALLOCATE and FREE statements or the ADDR built-in function may be used to access other storage.

- BEGIN

A statement that defines the beginning of a begin block.

- BINARY

An attribute specifying that the associated identifier has radix (base) two.

- BIT

An attribute specifying the associated identifier to be bit-string data.

- Bit String

A sequence of binary digits that can be operated on with the string functions.

- Bit String Data

A value whose data type is BIT or BIT ALIGNED.

- Blanks

A character used as a separator. Names and constants not otherwise separated must be separated by one or more blank characters. Additional blanks may be used around punctuation or between names or constants and punctuation, but are not required.

- Block

A program section of organized statements beginning with a PROCEDURE or BEGIN statement and ending with the matching END statement delimiting the scope of identifiers declared within the block without the EXTERNAL attribute. See external, internal blocks, procedure.

Activation of Begin Block: A begin block is activated when control passes to the BEGIN statement in a normal sequential manner.

Activation of a Procedure Block: A procedure block is activated when control is passed to it by a procedure reference [consider recursion] (a function reference or CALL statement).

Begin Block: An internal block starting with a BEGIN statement and ending with an END statement.

Dynamic Descendence of Blocks: Dynamic descendence is a term applicable to block relationships where a number of blocks are active simultaneously. Any block is a dynamic descendent of another block if it is activated from that block by a CALL statement, function reference, flow of control (for BEGIN blocks), or (on-unit) signal. A non-local GO TO passes control from block A to block B, thus deactivating block A and all Block B's dynamic descendents and leaving block B as the last extant active block.

External Block: A procedure block whose entry name is not within the scope of any (other) block. See procedure.

Immediate Dynamic Descendent Block: When a block (A) directly activates another block (B), block B is the immediate dynamic descendent of block A.

Internal Procedure Block: A procedure block whose entry name is within the scope of an encompassing block. See procedure.

Procedure Block: A block consisting of a series of statements beginning with a PROCEDURE statement and ending with an END statement.

Procedure Block Name: A name designating the entry point to a procedure block. (i.e., the name of the PROCEDURE statement).

Procedure Block Reference: Invocation of a procedure by CALL statement or function reference.

Recursive Procedure Block: A procedure block that calls itself or is called by one of its dynamic descendents.

Termination of Begin Block: A begin block is terminated by: (1) execution of the END statement for the block or (2) a non-local GO TO.

Termination of Procedure Block: A procedure block is terminated: (1) by execution of a RETURN statement or the END statement for the block; (2) by execution of a non-local GO TO statement.

- BUILTIN

An attribute specifying that the declared name is a built-in function.

- Built-in Function List

1. Arithmetic Built-in functions:

ABS	CEIL	DIVIDE	EXP	FLOOR
LOG	LOG2	LOG10	MAX	MIN
MOD	ROUND	SIGN	SQRT	TRUNC

2. Trigonometric Built-in functions:

ACOS	ASIN	ATAN	ATANH	ATAND	COS
COSD	COSH	SIN	SIND	SINH	
TAN	TAND	TANH			

3. String Built-in Functions:

BOOL	COLLATE	COPY	INDEX	LENGTH
STRING	SUBSTR		TRANSLATE	VALID
VERIFY				

4. Conversion Built-in functions:

BINARY	BIT	BYTE	CHARACTER
FIXED	FLOAT	RANK	DECIMAL

5. Condition Built-in functions:

ONCODE	ONFILE	ONKEY	ONLOC

6. Miscellaneous Built-in functions:

ADDR DATE DIMENSION HBOUND LBOUND LINENO

NULL PAGENO UNSPEC TIME

- Built-in Function Reference

A reference to a built-in function. Such a reference always produces the value of that function (it is never an entry value).

- CALL

A statement that creates the block activation of a procedure identified by the reference. The reference must be an entry name, entry variable, or entry-valued function.

- CHARACTER

An attribute specifying its associated identifier to be character-string data.

- Character String

A sequence of characters that can be operated on with the string functions.

- Character-String Data

A value whose data type is CHARACTER(n) or CHARACTER(n) VARYING.

- CLOSE

A statement that closes the file control block identified by the reference. The reference must identify a file name, file variable, or file-valued function.

- Comments

A non-executable remark added by the programmer. The general form of a comment is: /* Any sequence of characters except an asterisk followed immediately by a slash */. Comments may occur anywhere a blank could appear (i.e. between any two tokens).

- Compound Statement

Compound statements are statements that contain another statement. PL1G has two compound statements: the IF statement and the ON statement. See Section 9.

- Condition-names

Names that may be specified in PL1G ON or SIGNAL statements. The legal condition names are:

ENDFILE, ENDPAGE, ERROR, and KEY

- Constant

A sequence of characters that represents a particular value.

Arithmetic Constants: Arithmetic constants represent decimal values. (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).

Bit-String Constant: 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 octal and hexadecimal notation. For example:

'775'B3 /* octal notation */

'A70'B4 /* hexadecimal notation */

Character String Constant: Zero or more characters enclosed in single quotation marks. For example:

'98.6F'

Fixed-Point Constant: One or more numeric digits with optional sign and decimal point. For example:

7.5

Floating-Point Constant: One or more numeric digits with an optional decimal point. These digits are followed by the letter "E", in turn, followed by an optionally signed exponent representing an integral power of 10. For example:

3.12E-11

Integer Constant: One or more numeric digits. For example:

25

- Conversion

When a value is assigned to a variable, it is converted to the data type that has been declared for that variable.

- Data Type, Variable

A variable's data type determines what kind of values the variable can hold.

- Data Type Attribute

Data type attributes are:

FIXED BINARY

FIXED DECIMAL

FLOAT DECIMAL

FLOAT BINARY

PICTURE

CHARACTER

CHARACTER VARYING

BIT

BIT ALIGNED

POINTER

LABEL

ENTRY

FILE

- Data Type, Value

The data type of a value determines which operations can be performed on the value and how it can be represented in storage.

- Data Type, Constant

The data type of a constant is determined by the syntax of a constant. See constant.

- DECIMAL

An attribute specifying that the associated identifier has radix (base) 10.

- Declaration

The declaration of a name determines the meaning of the name. There are two kinds of declaration in PL1G: the DECLARE statement and the label prefix. See Section 5.

- Declared Names

A name used to denote an object operated upon by the program such as: a variable, a file, or a label.

- DECLARE Statement

A statement that clearly establishes the meaning of names. It is not an executable statement, and it may appear anywhere in a program except as part of a compound statement.

- DEFINED

An attribute that is used to specify an alternative description of another variable.

- DELETE

A statement that deletes a record from a KEYED SEQUENTIAL UPDATE file.

- DIMENSION

An attribute consisting of a list of bound pairs specifying the number of dimensions of an array, and the bounds of the dimension. This attribute also specifies that the declared name (identifier) is an array.

- DO

A statement delimiting the start of a DO-group and possibly specifying the manner of iteration of statements within the group. There are four kinds of DO statements: simple-do, do-while, do-repeat, and iterative-do.

- DO-group

A sequence of statements headed by a DO statement and ending with an END statement.

Simple-DO: A simple-DO statement execution causes the statements in the DO-group to be executed once and is used mainly for grouping statements used in THEN and ELSE clauses of IF statement.

DO-while:

DO WHILE (expression);

Execution of a DO-while statement causes the specified expression to be evaluated to produce a value that is a bit string of length one. If the bit string is TRUE (i.e. '1'B) the statements of the DO-group are executed, and when the corresponding END statement is executed, control is transferred back to reevaluate the specified expression and test for a new bit string value. If the bit string is FALSE (i.e. 'Ø'B) the statements in the DO-group are not executed and execution resumes with the statement following the corresponding END statement.

WHILE: A keyword separator that may appear in a DO statement. WHILE is followed by a parenthesized specification of conditions for iteration of the DO-group.

DO-repeat: DO index = start REPEAT next [WHILE (test)];

Iterative-DO: DO index = start [TO finish] [BY increment] [WHILE (test)];

BY: A keyword separator that may appear in a DO statement specifying the increment of the control variable for each iteration.

TO: A keyword separator that may appear in a DO statement specifying the limits of incrementation of the control variable.

Control Variable: A variable appearing in a DO statement that is assigned different values during each iteration of the DO-group. A control variable is also called an index variable.

DO-loop: An iterated DO-group.

DO Specification: The portion of a DO-repeat or an iterative-DO which is specified to the right of the equals sign.

- Edit Directed I/O

Transmission of data to or from a stream file under control of a format-list.

- END

A statement terminating DO groups, BEGIN blocks, and PROCEDURE blocks.

- Entry

An identifier associated with a procedure, by which reference may be made to the procedure. Procedures which are not part of the compiled module must be declared with the ENTRY attribute, and if they are functions, the RETURNS attribute. Entry data is used in PL1G to describe subroutines and functions that are not built in. ("ENTRY" is used instead of "PROCEDURE" because in full PL/I procedures can be entered at points other than the PROCEDURE statement.)

- ENTRY

An attribute specifying that the associated identifier is an entry constant or variable.

- Expression

An expression consists of operators and operands.

- Extents

Values that determine the variable's storage size; extents are evaluated when storage is allocated for the variable.

- EXTERNAL

An attribute specifying that all declarations of the associated identifier with the EXTERNAL attribute will share the same storage, allowing communication among blocks.

- File

An organized collection of data stored in the computer system primary or secondary storage. A file is referenced by using a file name declared with the FILE attribute.

Record File: A file organized into a set of discrete records that are either accessible sequentially or accessible directly by character-string valued keys.

Stream File: A file containing a sequence of characters organized into lines.

File Constant: A name declared with the FILE attribute. A file constant cannot be the target of an assignment statement.

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 (i.e., a file constant or the value of another file variable previously assigned a file constant).

File Value: A file value designates a file control block that can be opened or closed, and which, thereby, can 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.

- FILE

An attribute specifying the declared identifier (name) is a file, or if used with the VARIABLE attribute, the name is a file variable.

- FIXED

An attribute specifying the associated variable as representing a fixed-point number.

- FLOAT

An attribute specifying the associated variable as representing a floating-point number.

- FORMAT

A statement that defines a data format which may be referenced by a edit-directed GET or PUT statement.

- Format List

A format list controls the transmission of data to or from a stream I/O file during the execution of an edit-directed GET or PUT statement.

- Format-items

Elements of a format list, separated by commas.

- FREE

A statement releasing storage allocated for a based variable.

- Function Reference

A procedure reference that returns a value.

- GET

A statement that inputs (reads) information from a stream-I/O file.

- GO TO

A statement causing transfer of control.

Local GO TO: A GO TO statement transferring control within the block in which it appears.

Non-local GO TO: A GO TO statement transferring control to a statement in a dynamically encompassing block.

- Identifier

A string of alphanumeric and underscore (_) characters which begins with an alphabetic character. Identifiers are also called names.

- IF

A statement causing program flow to depend upon the truth value of an expression.

ELSE: An IF statement clause specifying alternative transfer of control should the necessary conditions of the IF statement not be met.

THEN: An IF statement clause specifying the path of control to be taken when the conditions of the IF statement are met.

- %INCLUDE

A PL1G statement that allows the insertion of the contents of a text file in the place of the %INCLUDE statement.

- INITIAL

An attribute specifying the initial value of a STATIC variable or member of a STATIC structure.

- INTERNAL

An attribute specifying its associated identifier to be limited to the scope of the block in which it is declared.

- Keyword

A name used to denote parts of statements such as: verbs, options, and clauses.

A keyword is an identifier that has specific meaning to the compiler in particular contexts.

- LABEL

An attribute specifying the associated identifier to be a statement label variable that can have different statement label constant or variable values assigned to it.

Label: An identifier associated with a statement by which reference may be made to that statement.

Label Prefix: An identifier, separated from its associated statement by a colon, enabling transfer of control to the associated statement by reference to its label prefix. A label prefix declares a name as the name of a format, as the name of a procedure, or as a statement label.

- Length

The number of characters in a character string value or the number of bits in a bit string value.

- Length Attribute

An attribute consisting of a parenthesized constant, expression or asterisk, specifying the maximum length of varying-length strings or the length of a fixed-length string. (See precision.)

- Level Number

An integer that specifies the position of the associated identifier within the hierarchical organization of a structure.

- List-Directed I/O

Transmission of data to or from a stream file without a format-list.

- Major Structure

A structure that is not a substructure (i.e., a level 1 structure).

- Member

An immediate component of a structure.

- Name

A string of alphanumeric and underscore (_) characters. A name is a sequence of up to 32 letters (both lower- and upper-case), and/or digits, and/or the underscore (_) and $ characters. The first character of a name must be a letter. Names are also called identifiers.

- Null Character String

A character string of zero length.

- Null Pointer

A pointer value produced by the NULL built-in function. A null pointer is a unique value that does not address any variable and is used to indicate that a pointer variable does not currently address anything.

- Null Statement

Null is an empty statement that has no effect.

- Null String

A zero length character- or bit-string.

- On-condition

The on-condition is a condition specified in an ON statement.

- ON

A statement specifying the action to be taken when a condition interrupt occurs for the condition named in the statement.

- 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.

- OPEN

A statement that causes the specified file control block to be opened with the line size, page size, and attributes specified in the OPEN statement.

- Operand

A part of an expression. It may be: a constant, a variable reference, a function reference, a built-in function reference, or another expression.

- Operators

The operators used in PL1G are: + - * / ** = ^= < > <= >= ^< ^> & ^ | (or !) and || (or !!).

Arithmetic Operators: The arithmetic operators are: +, denoting addition or prefix positive; -, denoting subtraction or prefix negative; *, denoting multiplication: /, denoting division, and **, denoting exponentiation.

Bit-String Operators: The bit string operators are: ^, denoting NOT; &, denoting AND; and | (or !), denoting OR.

Comparison Operators: The comparison operators are: >, denoting greater than; ^>, denoting not greater than; >=, denoting greater than or equal to; =, denoting equal to; ^=, denoting not equal to; <=, denoting less than or equal to; <, denoting less than, and ^<, denoting not less than.

Infix Operators: Operators placed between expressions.

Prefix Operators: The operators +, -, and ^, placed to the left of expressions.

String Operator: The string operator is || (or !!), denoting concatenation.

- Parameter

An identifier in a PROCEDURE statement for which a value is substituted by an invoking reference.

- PICTURE

An attribute that specifies pictured values for the declared name. The attribute declaration contains an image of the data and specifies the editing to be performed each time that a value is assigned to the variable.

- PICTURED Data

A value whose data type is determined by the PICTURE attribute or the P-format in a FORMAT list.

- POINTER

An attribute specifying the declared identifier as a pointer variable that contains the address of a based storage datum. Used with ADDR, ALLOCATE, and references to BASED variables.

Pointer Data: A value whose data type is POINTER. This pointer value is the address of a variable's storage.

Pointer Qualification: Identification of a generation of a based variable through use of a pointer followed by the pointer qualification symbol, e.g., Q -> ALPHA. Pointer-valued functions and pointer-valued built-in functions may also be used as pointer qualifiers.

Pointer Qualification Symbol: The character sequence -> signifies that the pointer value on the left of -> gives the address to be used with the BASED variable reference on the right.

Pointer Variable: A pointer is a variable capable of holding the address of any PL1G variable. It must be declared with the POINTER data type attribute.

- Precision

The number of significant binary or decimal digits maintained for the value of an arithmetic variable and, optionally, how many of those digits are fractional. As an attribute it is specified by a parenthesized decimal number or pair of numbers separated by a comma which is the number of significant binary or decimal digits to be maintained for either a fixed-point or floating-point datum and, optionally (for fixed point data only), the number of those digits which are fractional.

- Procedure

A sequence of statements beginning with a PROCEDURE statement and terminated by the matching END statement. A procedure defines a block of statements and is sometimes called a block.

External Procedure: A procedure not contained within another procedure.

Internal Procedure: A procedure that is contained within another procedure.

Nested Procedure: See internal procedure

- Procedure Reference

Any reference 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.)

- Program

The single result of compiling, loading, and executing one or more program modules.

- Program Module

The program text that is input to the compiler.

- Pseudo-Variable

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 pseudo-variables are PAGENO, STRING, SUBSTR, and UNSPEC.

- Punctuation

Punctuation symbols are either operators or separators.

- PUT

A statement that outputs (writes) information to a stream-I/O file.

- READ

A statement that causes a record file to be read.

- Recursive procedure

A procedure that can call itself. The notion of recursive 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:

```
N! = 1 for N=1
N! = N * (N-1)! for N>1
```

- Reference

Use of a name in a context other than in a declaration.

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.

A reference is 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.

Symbol Reference: A name without any subscripts, pointer reference, or argument list.

Subscripted Reference: A name that has been declared as an array, followed by a parenthesized list of subscript expressions.

- %REPLACE

A PL1G extension statement that allows the replacement of each subsequent occurrence of a name by an optionally-signed constant.

- Reserved Names

There are no reserved names (i.e., reserved words) in PL1G.

- RETURN

A statement that terminates the current procedure activation and returns control to the calling block.

- RETURNS

An attribute associated with an explicitly declared entry name or an option in a procedure statement. RETURNS is followed by a parenthesized list defining the data attributes of the value to be returned by the entry invoked as a function.

- REWRITE

A statement that replaces a record in a record file that has the KEYED UPDATE attributes.

- Scale

A scale is either fixed-point (FIXED) or floating-point (FLOAT).

Fixed-point Scale: That format of arithmetic data in which the datum is a rational binary or decimal number with specified number of digits.

Floating-point Scale: That format of an arithmetic datum in which the datum is a rational number with a fractional part and an exponent part.

- Scope of a Name

The region of a program over which a name (identifier) is known. The scope of a name is that region of the program in which the name is referenceable. 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.

- Scope of Declaration

The scope of a declaration includes the block in which it appears and all contained blocks, except blocks in which the name (identifier) has been redeclared.

- Separators

Separators are the following characters: ( ) , . ; : and the blank character.

- SIGNAL

A statement that signals a specified condition. (Normally used during debugging to test on-units).

- 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.

- STATIC

An attribute specifying that storage for the associated variable is allocated at the start of program execution, and released when the program terminates.

- Statement

A statement is a sequence of tokens ending with a semicolon. All statements, except the assignment statement, begin with a keyword that identifies the purpose of the statement.

Statement Identifier: A keyword naming a statement, e.g., "DO" is the statement identifier of the DO statement.

Statement Label: A name identifying a statement.

- STOP

A statement which closes all open files and terminates program execution.

- Storage

Allocation of Storage: Association of a specified region of storage with a variable.

Automatic Storage: Storage allocated for a variable when the block in which it is declared is activated and then released when that block is terminated.

Based Storage: Storage of variables always referenced with a pointer value that indicates the generation of the variable.

Generation: A single copy of data, representing a specific allocation of a given variable.

Static Storage: Storage allocated before execution of the program and released at program termination.

Storage Class: A variable's storage class determines how and when storage is allocated for the variable.

Storage Class Attribute: The storage class attributes are:

AUTOMATIC

STATIC

BASED

DEFINED

- String

A sequence of bits or characters on which string operations are allowed.

- Structure

A hierarchically ordered set of variables that may be of different data types.

- Structure Qualified Reference

A sequence of names written left to right in order of increasing level numbers. The names are separated by periods. Optionally, blanks may be inserted around the periods. The sequence must include sufficient names to make the reference unique.

Fully Qualified Reference: A structure qualified reference that includes the name of each containing structure from the major structure down to the referenced member.

Partially Qualified Reference: A structure qualified reference that is unique, but one or more of the names of containing structures have been omitted.

- Subroutine Procedure

A procedure invoked by a subroutine reference is called a subroutine procedure or simply a subroutine. Subroutines can return indirectly one or more values to the calling procedure through shared variables or its parameters or may return no values.

- Subroutine Reference

A procedure reference following the keyword CALL is called a subroutine reference. It does not return a value.

- Subscript

An integer, or integer-valued expression, used to reference an array element. Elements of an array are referenced using as many subscripts as the array has dimensions.

- Substructure

A structure that is itself a member of another structure.

- Token

The basic elements of the PL1G language. A token is: a name, a constant, a punctuation symbol, a comment, or a compile-time text modification statement.

- Variable

A variable is a named object that is capable of holding values. Each variable has two properties: data type and storage class.

- VARIABLE

A keyword specifying part of a file or entry data type attribute. VARIABLE specifies that the declared name is a file variable or entry variable.

- VARYING

An attribute specifying its associated identifier to be a varying-length string.

APPENDIX B

ABBREVIATIONS

Abbreviations are provided for certain keywords and builtin function names. The abbreviations will be recognized as synonymous in every respect with the full denotations, except that in the case of builtin-function-names the abbreviations have separate declarations (explicit or contextual) and name scopes. The abbreviations are shown to the right of the full denotations in the following list.

ALLOCATE	ALLOC
AUTOMATIC	AUTO
BINARY	BIN
CHARACTER	CHAR
COLUMN	COL
DECIMAL	DEC
DECLARE	DCL
DEFINED	DEF
DIMENSION	DIM
EXTERNAL	EXT
INITIAL	INIT
INTERNAL	INT
PICTURE	PIC
POINTER	PTR
PROCEDURE	PROC
SEQUENTIAL	SEQL
VARYING	VAR

APPENDIX C

DATA FORMATS

OVERVIEW

The PL1G language supports the following data types:

FIXED BINARY
FIXED DECIMAL
FLOAT BINARY
FLOAT DECIMAL
PICTURE

CHARACTER
CHARACTER VARYING
BIT
BIT ALIGNED

POINTER

LABEL
ENTRY

FILE

These data types are described in Section 3. The following provides a series of descriptions showing how the data is internally represented in storage, and gives 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 twos-complement binary number.

Precision: 1 <= P <= 31

Default Precision: 15

Alignment: Word.

Storage Requirements: 1 <= P <= 15 One word.
16 <= P <= 31 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 4-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 <= P <= 14

Default Precision: 5

Alignment: Byte.

Storage Requirements: FLOOR((P+2)/2) bytes.

Internal Representation

Each nybble holds one decimal digit. The last nybble holds an indicator of the sign.

FLOAT BINARY DATA

Precision: 1 <= P <= 47

Default precision: 23

Alignment: Word.

Storage Requirement: 1 <= P <= 23 2 words.
24 <= P <= 47 4 words.

Internal Representation

Precision 1-23: Two words.

Bit 1:	Sign
Bits 2-24:	Mantissa
Bits 25-32:	Exponent

Precision 24-47: Four words.

Bit 1:	Sign
Bits 2-48:	Mantissa
Bits 49-64:	Exponent

FLOAT DECIMAL DATA

Precision: 1 <= P <= 14

Default Precision: 6

Alignment: Word.

Storage Requirement: 1 <= P <= 6 2 words.
7 <= P <= 14 4 words

Internal Representation

Precision 1-6: Two words.

Bit 1: Sign
Bits 2-24: Mantissa
Bits 25-32: Exponent

Precision 7-14: Two words.

Bit 1: Sign
Bits 2-48: Mantissa
Bits 49-64: Exponent

PICTURE DATA

Values to be assigned to a pictured variable are first converted to a decimal value according to the normal PL1G 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, which is placed in the procedure section by the compiler before any generated code. The edit subprogram is part of the following structure:

```
DCL 1 PICTURE_INFO,
      2 IGNORE FIXED BIN (31),
      2 SCALE_FACTOR BIT (8),
      2 NUMBER_OF_DIGITS BIT (8),
      2 TYPE_OF_VALUE BIT (8),
      2 SUBPROGRAM_SIZE BIT (8),
      2 CHAR_SIZE (BIT (8),
      2 EXP_INDEX BIT (8),
      2 EXP_NUMBER_OF_DIGITS BIT (8),
      2 BLANK_MEANS_NEGATIVE BIT (1),
      2 RESERVED BIT (7),
      2 SUBPROGRAM CHAR(PICTURE_INFO.SUBPROGRAM_SIZE);
```

Picture data is byte-aligned and requires n bytes of storage, where n is the number of picture characters excluding any V character.

Note

BIN (NUMBER_OF_DIGITS) and BIN (SCALE_FACTOR) give the precision of the FIXED DECIMAL value described by the picture (e.g., PICTURE '99,999V.99' gives a NUMBER_OF_DIGITS of 7 and a SCALE_FACTOR of 2).

BIN (SUBPROGRAM_SIZE) gives the number of bytes in the edit subprogram.

BIN (CHAR_SIZE) gives the number of characters in the picture value.

BLANK_MEANS_NEGATIVE is '1'B if the picture has the "+" sign character.

SUBPROGRAM is the edit subprogram for XED.

TYPE_OF_VALUE, EXP_INDEX, and EXP_NUMBER_OF_DIGITS are used to support full PL/I features which are not available in Subset G.

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: 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: 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.

POINTER DATA

Alignment: Word.

Storage Requirement: 3 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 (iff bit 4 is set)
Bits 37-48:	Reserved

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 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: 4 words.

Internal Representation

Four words are used.
First two words: Address of executable statement.

Bits 1-16: Word number
Bit 17: Fault code
Bits 18-19: Ring number
Bit 2Ø: Data format indicator (always Ø)
Bits 21-32: Segment number

Second two words: Address of target stack frame.

Bit 33: Fault code
Bits 34-35: Ring number
Bit 36: Data format indicator (always Ø)
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 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: 4 words.

Internal Representation

Four words are used.
First two words: ECB address.

Bit 1:	Fault code
Bits 2-3:	Ring number
Bit 4:	Data format indicator (always Ø)
Bits 5-16:	Segment number
Bits 17-32:	Word number

Second two words: Display pointer.

Bit 33:	Fault code
Bits 34-35:	Ring number
Bit 36:	Data format indicator (always Ø)
Bits 37-48:	Segment number
Bits 49-64:	Word number

FILE DATA

At Rev. 17, PL1G SEQUENTIAL files are in standard RDBIN/WRBIN format, and STREAM files are in standard RDASC/WRASC format. DIRECT files are supported using PRWF$$ to position to the appropriate word in the file, which is calculated as:

(KEYVALUE * 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 Ø)
Bits 5-16:	Segment number
Bits 17-32:	Word number

APPENDIX D

STACK FRAME AND FUNCTION RETURN CONVENTIONS

LOCATIONS OF RETURNED FUNCTION VALUES

Returns Type	Where Returned	
	V-mode	I-mode
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
ptr	FARØ	FARØ

For all other data types, the calling procedure sets up FARØ 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 FARØ to SB%+4Ø to SB%+42 of its stack frame.

STACK FRAME FORMAT

Figure D-1 shows a typical stack frame format for a PL1G application. This figure is followed by a series of notes that explain the stack frame format entries.

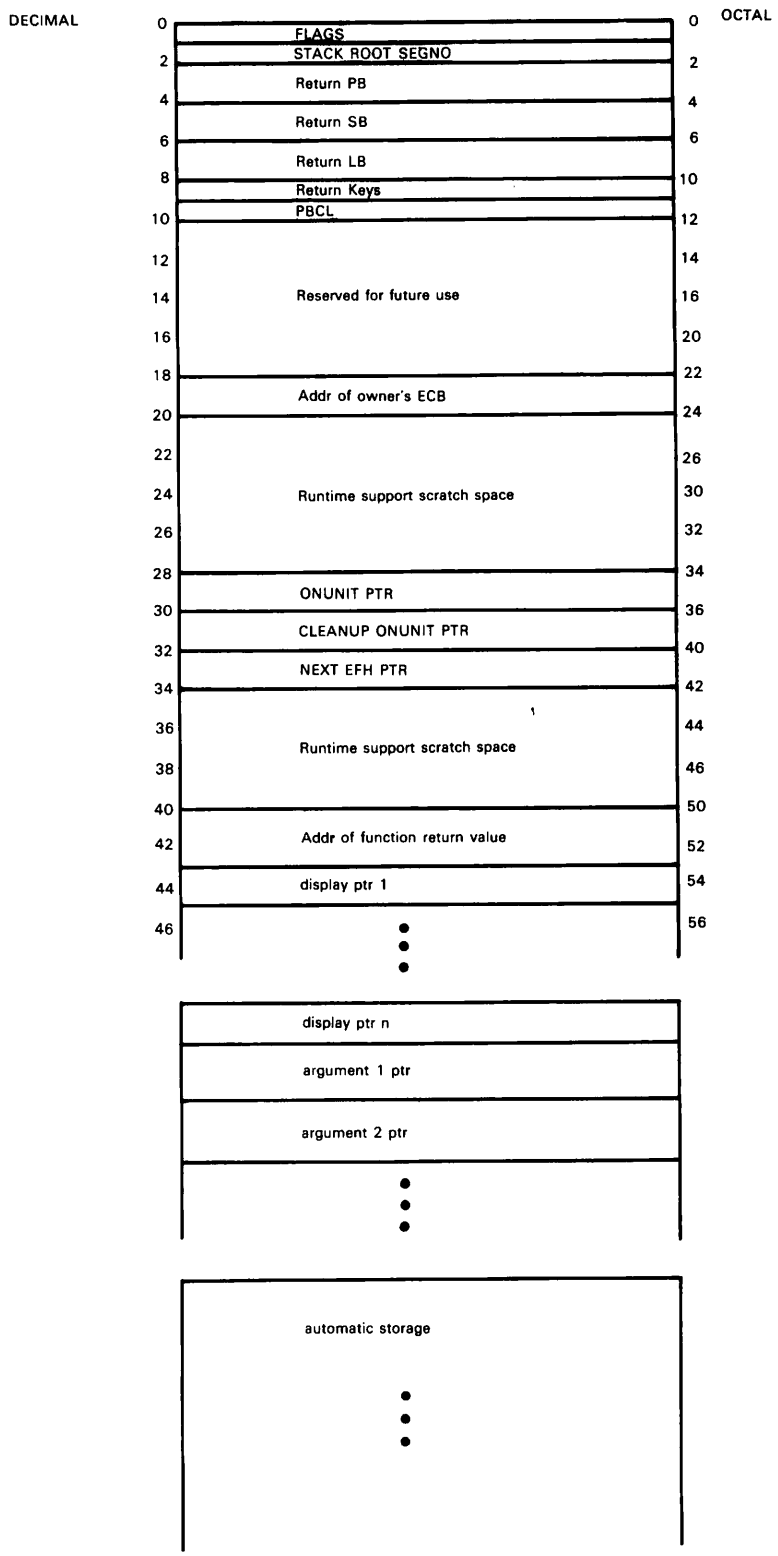

Figure D-1. Stack Frame Format

Notes on stack frame format:

- Bit 5 of FLAGS will be set for PL1G procedure stack frames.
- SB%+1 to SB%+9 (all offsets referenced in decimal) is the hardware-defined portion of the stack.
- SB%+18 points to the ECB (Entry Control Block) of the owning PL1G block, for both procedure and begin blocks. The ECB of a PL1G procedure block is immediately followed by a char(*)var giving the name of the procedure/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 will be 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 - may be 0) 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 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.

APPENDIX E

ASCII CHARACTER SET

The standard character set used by Prime is the ANSI, ASCII 7-bit set.

PRIME USAGE

Prime hardware and software uses standard ASCII for communications with devices. The following points are particularly important to Prime usage.

- Output Parity is normally transmitted as a zero (space) unless the device requires otherwise, in which case software will compute transmitted parity. Some controllers (e.g., MLC) may have hardware to assist in parity generations.
- Input Parity is ignored by hardware and by standard software. Input drivers are responsible for making the parity bit suit the host software requirements. Some controllers (e.g., MLC) may assist in parity error detection.
- The Prime internal standard for the parity bit is one, i.e., '200 is added to the octal value.

KEYBOARD INPUT

Non-printing characters may be entered into text via the editor by giving the logical escape character (^ by default) followed by the octal value of the character. The character is interpreted by output devices according to their hardware.

Example: Typing ^207 will enter one character into the text.

CTRL-P	('220)	is interpreted as a .BREAK.
.CR.	('215)	is interpreted as a newline (.NL.)
"	('242)	is interpreted as a character erase
?	('277)	is interpreted as line kill
\	('334)	is interpreted as a logical tab (Editor)

CHANGING THE SIGNIFICANCE OF SPECIAL CHARACTERS

When a character having a special meaning to the Prime system is needed for some other purpose, that meaning can be transferred to another character with the editor's SYMBOL command (see The New User's Guide to Editor and Runoff) or with the PRIMOS TERM command (see The PRIMOS Commands Reference Guide). The original special character is thereby freed for ordinary use.

Table E-1

ASCII Character Set (Non-Printing)

Octal Value	ASCII Char	Comments/Prime Usage	Control Char
200	NULL	Null character - filler	^@
201	SOH	Start of header (communications)	^A
202	STX	Start of text (communications)	^B
203	ETX	End of text communications	^C
204	EOT	End of transmission (communications)	^D
205	ENQ	End of I.D. (communications)	^E
206	ACK	Acknowledge affirmative (communications)	^F
207	BEL	Audible alarm (bell)	^G
210	BS	Back space one position (carriage control)	^H
211	HT	Physical horizontal tab	^I
212	LF	Line feed; ignored as terminal input	^J
213	VT	Physical vertical tab (carriage control)	^K
214	FF	Form feed (carriage control)	^L
215	CR	Carriage return (carriage control) (1)	^M
216	SO	RRS-red ribbon shift	^N
217	SI	BRS-black ribbon shift	^O
220	DLE	RCP-relative copy (2)	^P
221	DC1	RHT-relative horizontal tab (3)	^Q
222	DC2	HLF-half line feed forward (carriage control)	^R
223	DC3	RVT-relative vertical tab (4)	^S
224	DC4	HLR-half line feed reverse (carriage control)	^T
225	NAK	Negative acknowledgement (communications)	^U
226	SYN	Synchronocity (communications)	^V
227	ETB	End of transmission block (communications)	^W
230	CAN	Cancel	^X
231	EM	End of Medium	^Y
232	SUB	Substitute	^Z
233	ESC	Escape	^[
234	FS	File separator	^\
235	GS	Group separator	^]
236	RS	Record separator	^^
237	US	Unit separator	^_

Notes for Table E-1

1. Interpreted as .NL. at the terminal.

2. .BREAK. at terminal. Relative copy in file; next byte specifies number of bytes to copy from corresponding position of preceding line.

3. Next byte specifies number of spaces to insert.

4. Next byte specifies number of lines to insert.

Conforms to ANSI X3.4-1968

The parity bit ('2ØØ) has been added for Prime-usage.

Non-printing characters (^c) can be entered at most terminals by typing the (control) key and the c character key simultaneously.

Table E-2

ASCII Character Set (Printing)

Octal Value	ASCII Character	OCTAL Value	ASCII CHaracter	OCTAL Value	ASCII Character
240	.SP (1)	300	@	340	` (9)
241	!	301	A	341	a
242	" (2)	302	B	342	b
243	# (3)	303	C	343	c
244	$	304	D	344	d
245	%	305	E	345	e
246	&	306	F	346	f
247	' (4)	307	G	347	g
250	(	310	H	350	h
251	)	311	I	351	i
252	*	312	J	352	j
253	+	313	K	353	k
254	, (5)	314	L	354	l
255	-	315	M	355	m
256	.	316	N	356	n
257	/	317	O	357	o
260	0	320	P	360	p
261	1	321	Q	361	q
262	2	322	R	362	r
263	3	323	S	363	s
264	4	324	T	364	t
265	5	325	U	365	u
266	6	326	V	366	v
267	7	327	W	367	w
270	8	330	X	370	x
271	9	331	Y	371	y
272	:	332	Z	372	z
273	;	333	[	373	{
274	<	334	\	374	|
275	=	335	]	375	}
276	>	336	^ (7)	376	~ (10)
277	? (6)	337	_ (8)	377	DEL (11)

Notes for Table E-2

1. Space forward one position
2. Default terminal usage - erase previous character
3. British pound sign in British use
4. Apostrophe/single quote
5. Comma
6. Default terminal usage - kill line
7. 1963 standard ↑. Default editor use - logical escape
8. 1963 standard ←
9. Grave
10. 1963 standard ESC
11. Rubout - Not used for erase function by the Prime system

Conforms to ANSI X3.4-1968
1963 variances are noted

The parity bit ('200) has been added for Prime usage.

APPENDIX F

DIFFERENCES BETWEEN FULL PL/I AND PL/I SUBSET G

FEATURES SUPPORTED IN PL/I SUBSET G

For a list of the features in PL/I Subset G, scan the index entries under the headings: Attributes, Built-in functions, Conditions, Data types, Declarations, Files, Input/Output, Options, Statements, and Storage classes. The few differences between PL/I Subset G and PL1G are listed under "THE PL1G LANGUAGE" in Section 1.

FEATURES NOT SUPPORTED IN PL/I SUBSET G

The following features of full PL/I are not supported in PL/I Subset G. Features marked with a star (*) are non-standard extensions to ANS PL/I.

Program Elements

- The IBM 48-character set* is not supported.
- Blanks are not permitted to appear within compound operators* such as <=, >=, ^=, ^<, ^>, ||, ->, and **.
- Multiple label prefixes are not permitted on any statement.
- A DECLARE statement may not have a label prefix.
- A label prefix is permitted to have only one subscript.
- The use of subscripted label prefixes on PROCEDURE and FORMAT statements is not supported.
- The use of a label prefix on the THEN or ELSE clause of an IF statement is not supported.
- Condition prefixes are not supported.

Data Elements

- Binary constants are not supported.
- Replication factors are not supported.
- Sterling constants* are not supported.

- The use of the default-suppressing character P or the explicit-scale-factor character F in arithmetic constants is not supported.

- The use of a *-length string returns description in a RETURNS attribute or RETURNS option of a PROCEDURE statement is not supported.

- The use of functions which return array or structure values is not supported.

- Array bounds and string length specifiers for variables which are declared STATIC, or are members of a STATIC structure, must be integer constants.

- Array bounds and string length specifiers for variables which are parameters, or are members of a parameter structure, must be asterisks or integer constants.

Program Structure

- The OPTIONS option of the BEGIN statement is not supported.

- Multiple closure is not supported. Any closure label on an END statement must match the label prefix of the corresponding PROCEDURE, BEGIN, or DO statement.

- The following options of the PROCEDURE and ENTRY statements are not supported:

  ORDER*, IRREDUCIBLE*, RECURSIVE*, REDUCIBLE*, REORDER*

- The following options of the OPTIONS option of the PROCEDURE and ENTRY statements are not supported:

  COBOL*, FORTRAN*, NOMAP*, NOMAPIN*, NOMAPOUT*, REENTRANT*, TASK*

Declarations and Attributes

- The following attributes are not supported:

  AREA, BACKWARDS*, BUFFERED*, COMPLEX, CONDITION, CONNECTED*, CONTROLLED, EVENT*, EXCLUSIVE*, FORMAT, GENERIC, IRREDUCIBLE*, LIKE, LOCAL, OFFSET, POSITION, REDUCIBLE*, TASK*, TRANSIENT*, UNBUFFERED*

- The following attributes are supported, but are not keywords in the subset language:

  CONSTANT, DIMENSION, MEMBER, NONVARYING, PRECISION, PARAMETER, REAL, STRUCTURE, UNALIGNED

- The INITIAL attribute may be applied only to variables whose storage class is STATIC. Variables whose storage class is AUTOMATIC must be initialized via assignment statements. The INITIAL attribute may contain only string or arithmetic constants, or the NULL built-in function as initial values. The CALL option* of the INITIAL attribute is not supported.

- A nonzero scale factor may be specified only for a FIXED DECIMAL variable, and a negative scale factor may not be specified.

- BIT VARYING is not supported.

- iSUB defining is not supported.

- The allowable forms of pictures are restricted to a subset of the fixed-point decimal pictures. Repetition factors (e.g., PICTURE '(8)9') are not permitted. Sterling picture data is not supported.

- The following PICTURE characters are not supported:

  A, E, G*, H*, I, K, M*, P*, R, T, X, Y, 6*, 7*, 8*

- All names used in a program must be explicitly declared, either in a DECLARE statement, or else by use of the name as a label prefix or built-in function.

- The arithmetic default attributes are always FIXED BINARY, rather than FIXED DECIMAL or FLOAT DECIMAL according to the IJKLMN rule*. In particular, FIXED implies FIXED BINARY, so that FIXED(8,3) must be written as FIXED DECIMAL(8,3).

- Subscripted labels cannot be declared in a DECLARE statement. (In IBM PL/I, they must be*.)

- The ENTRY declaration for an EXTERNAL procedure may not be omitted*, as contextual declaration of an EXTERNAL procedure by its appearaance in a CALL statement is not supported.

- The parameter descriptor list in an ENTRY attribute may not be omitted. This feature is sometimes used to suppress the conversion of arguments which do not agree in type or shape with the corresponding parameters.

- The ALIGNED attribute may be specified only for CHARACTER and BIT variables. It may not be specified for a structure.

- The interpretation of the ENVIRONMENT attribute is implementation-defined, and subject to variation. The following options of the ENVIRONMENT attribute 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*, CTLASA*, CTL360*, COBOL*, INDEXAREA()*, NOWRITE*, ADDBUFF*, GENKEY*, NCP()*, TRKOFL*, SCALARVARYING*, KEYLENGTH()*, KEYLOC()*, ASCII*, BUFOFF()*, PASSWORD()*

- The interpretation of the OPTIONS attribute is implementation-defined, and subject to variation. The following options of the OPTIONS attribute are not supported.

  ASSEMBLER*, COBOL*, FORTRAN*, INTER*, NOMAP*, NOMAPIN*, NOMAPOUT*, RETCODE*

- A FILE CONSTANT may not be dimensioned.

- The DEFAULT statement is not supported. Note that the VALUE and DESCRIPTORS options* of the DEFAULT statement are not included in ANS PL/I.

Expressions, Conversions, and Assignment

- Expressions which produce aggregate values are not supported. All operators and function references must yield scalar values.

- Aggregate promotion is not supported, except for one case. The only allowed form of aggregate promotion is an assignment statement of the form "<array reference> = <scalar expression>", with the restriction that the <array reference> must denote connected storage. Aggregate assignments are otherwise restricted to the form "<aggregate reference 1> = <aggregate reference 2>", where both references must denote connected storage and possess identical descriptions as to type and shape. Note that the rules for the promotion of one aggregate type to another differ between IBM PL/I and ANS PL/I*.

- Implicit conversions between arithmetic or pictured data, bit-string data, and character-string data is not supported.

- The multiple target form of the assignment is not supported.

- BY NAME assignment is not supported.

- Overlapping string assignment is restricted such that, if the source and target strings overlap, the target string must begin to the left of the source string, and data movement must be from right to left.

Built-in Functions and Pseudo-variables

- The following built-in functions are not supported:

  ADD, AFTER, ALL*, ALLOCATION, ANY*, BEFORE, COMPILETIME*, COMPLETION*, COMPLEX, CONJG, COUNT*, COUNTER*, CURRENTSTORAGE*, DATAFIELD*, DECAT, DOT, EMPTY, ERF, ERFC, EVERY, HIGH, IMAG, LOW, MULTIPLY, NULLO*, OFFSET, ONCHAR, ONCOUNT*, ONFIELD, ONSOURCE, PARMSET*, PLIRETV*, POINTER, POLY*, PRECISION, PRIORITY*, PROD, REAL, REPEAT*, REVERSE, SAMEKEY*, SOME, STATUS*, STORAGE*, SUBTRACT, SUM

- The following pseudo-variables are not supported:

  COMPLEX*, IMAG, ONCHAR, ONSOURCE, REAL

- Pseudo-variables may appear only on the left-hand-side of an assignment statement.

- The UNSPEC built-in function requires that the argument be a reference. In IBM PL/I, the argument may be any expression*.

- The MAX and MIN built-in functions must have exactly two arguments.

- The FIXED and FLOAT built-in functions require the second argument denoting the precision of the result.

- Various built-in functions have restrictions on their arguments in order to prevent the creation of unsupported data types.

Storage Control

- The ALLOCATE and FREE statements may allocate or free only one item.

- The SET option is required in the ALLOCATE statement, since CONTROLLED storage is not supported.

- The IN option of the ALLOCATE statement is not supported.

Program Control

- The comma-list form of the DO statement is not supported.

- The UNTIL clause* of the DO statement is not supported.

- The following statements are not supported:

  EXIT*, HALT*, LEAVE*, SELECT*

Conditions and Exception Control

- The following conditions are not supported:

  Programmer-named conditions, AREA, ATTENTION*, CHECK*, CONVERSION, FINISH, NAME, PENDING*, RECORD, SIZE, STORAGE, STRINGRANGE, STRINGSIZE, SUBSCRIPTRANGE, TRANSMIT

- Condition prefixes are not supported.

- An ON or REVERT statement may specify only one condition.

- A RETURN statement may not be used within an on-unit.

- The SNAP and SYSTEM options of the ON statement are not supported.

Functions and Procedures

- A function or procedure reference may have only one argument list.

- A reference to a function with no arguments returns an ENTRY value. An empty argument list is required to cause invocation of a parameterless function. In IBM PL/I, a reference to a function with no arguments will cause 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 use of the ENVIRONMENT file attribute in Prime PL/I Subset G is not generally compatible with its use in other implementations.

- The following I/O statements are not supported:

  DISPLAY*, LOCATE, UNLOCK*

- An OPEN or CLOSE statement may specify only one file.

- The TAB option of the OPEN statement is not supported.

- The ENVIRONMENT, LEAVE*, REREAD*, and REWIND* options of the CLOSE statement are not supported.

- The use of the TITLE option is generally not compatible with its use in other implementations.

- GET and PUT statements may contain only one I/O list, and at most one format list.
- The DATA option of the GET and PUT statements is not supported. Consequently, data-directed I/O is not supported.
- The COPY option of the GET statement is not supported.

  The SNAP*, FLOW*, and ALL* options of the PUT statement are not supported.
- A LINE option may not be specified as part of a PAGE option*.
- The INTO and FROM options of the READ, WRITE, REWRITE, and DELETE statments may not reference bit-aligned items.
- The FROM option is required in the REWRITE statement.
- The IGNORE option of the READ statement is not supported.
- The EVENT option* of the READ and WRITE statements is not supported.
- The use of expressions and variable references in format lists is not supported.
- The specification of the external precision in E and F format items by means of a third parameter is not supported.
- The C format item is not supported.

Preprocessor Facilities

- The preprocessor facilities* of IBM PL/I are not supported. This includes 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*, %PAGE*, %SKIP*

Multitasking Facilities

- The multitasking facilities* of IBM PL/I are not supported. This includes 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 pseudo-variables.

Diagnostic Facilities

- The diagnostic facilities* of IBM PL/I are not supported. This includes the following features:

  The CHECK condition prefix* and the CHECK condition*.

  The CHECK*, FLOW*, NOCHECK*, and NOFLOW* statements.