DIGITAL RESEARCH™

PL/I™ Language Reference Manual

DIGITAL RESEARCH™

PL/I Language Reference Manual

The *PL/I Language Reference Manual* was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press/ Monterey.

First Edition: October 1982

Foreword

The PL/I system is a complete software package for both applications and system programming. Digital Research has implemented PL/I for both the 8080 and 8086 processors. These implementations are compatible at the source code level. This manual describes source code requirements for both implementations.

PL/I runs under the Digital Research single-user operating systems, CP/M®, CP/M-86®, or concurrent CP/M-86™. It runs in a multi-user environment under MP/M II™ or MP/M-86™. This manual assumes you are already familiar with the operating system you are using and minimizes references to specific operating systems.

The *PL/I Language Reference Manual* is the formal specification of the PL/I programming language. This manual is primarily intended to be a reference document and is therefore not tutorial in nature. Some previous programming experience with PL/I or with another language is assumed.

The *Language Reference Manual* describes the overall structure and organization of PL/I source programs in the form of blocks and procedures. There is also a specification of the character set of the language, rules governing the formation of identifiers, constants, delimiters, operators, and comments.

This manual describes the various data types allowed in PL/I, including arrays and structures, as well as the rules governing conversion between data types. This manual also describes rules governing the scope of data declarations.

Assignments and expressions, sequence control statements, run-time storage management, and I/O processing are also described.

Finally, the manual describes all of the PL/I built-in functions including arithmetic, mathematical, string, conversion, condition, and miscellaneous functions.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

List of Figures

Table of Contents (continued)

List of Tables

Section 1 Introduction

PL/I is an implementation of PL/I for microcomputers that use the 8080, 8086, 8088 or similar processor. It is formally based on the ANSI General Purpose Subset (Subset G) as specified by the ANSI PL/I Standardization Committee X3J1. Subset G has the formal structure of the full language, but in some ways it is a new language and in many ways an improved language compared to its parent.

PL/I Subset G is easier to learn and use, and it is a highly portable language because its design generally insures hardware independence. It is also more efficient and cost effective as programs written in PL/I Subset G are easier to implement, document, and maintain. In many ways PL/I Subset G can be considered the first high-level, standardized, general-purpose programming language.

1.1 Documentation Set

The *PL/I Language Reference Manual* presents a detailed but concise description of the PL/I programming language. It is not a tutorial on how to program in PL/I, but rather a functional description of the language, its syntax, and semantics. This manual is a reference document that supplements Digital Research's *PL/I Language Programming Guide.*

The *PL/I Language Programming Guide* includes example programs that illustrate many of the features of PL/I, as well as the mechanical aspects of compiling and linking programs. If you have not programmed in PL/I before, you should read the *Programming Guide* first, while cross-referencing specific topics in the *Reference Manual.* If you are already an experienced PL/I programmer, you might want to read the *Reference Manual* only.

The *PL/I Language Command Summary* lists all the PL/I keywords and statement forms, data attributes, and error messages. It also contains a summary of the commands for the Compiler.

1.2 Notation

The following notational conventions appear throughout this document:

- Words in capital letters are PL/I keywords.
- Words in lower-case letters or in a combination of lower-case letters and digits separated by a hyphen represent variable information for you to select. These words are described or defined more explicitly in the text.
- Example statements are given in lower-case.
- the vertical bar | indicates alternatives.
- ƀ represents a blank character.
- Square brackets [ ] enclose options.
- Ellipses . . . indicate that the immediately preceding item can occur once, or any number of times in succession.
- Except for the special characters listed above, all other punctuation and special characters represent the actual occurrence of those characters.
- Within the text, the symbol CTRL represents a control character. Thus, CTRL-C means control-C. In a PL/I source program listing or any listing that shows example console interaction, the symbol ^ represents a control character.
- The acronym BIF refers to one of the PL/I built-in functions.

End of Section 1

Section 2 Program Structure

2.1 High-level Organization

Every PL/I program is composed of statements from the following general categories:

- Structural statements
- Declarative statements
- Executable statements

Structural statements define distinct, logical units within a program and therefore determine the overall, high-level organization. When a program executes, control always flows from one of these logical units to another. Logical units can contain other logical units; they can be nested. Structural statements also determine the hierarchical structure of a program where some logical units are subordinate to others.

Declarative statements determine the environment of a logical unit. The environment is simply the names and attributes of variables that are available or active in a logical unit. Declarative statements specify the context of variables that can be legally manipulated in a logical unit.

Executable statements are statements that perform some action. Both structural statements and declarative statements serve only to create a context for executable statements. All executable statements fall into one of the following categories:

- Preprocessor statements that execute at compile time and manipulate external source files.
- Assignment statements that assign the value of an expression or constant to a variable.
- I/O statements that control the flow of data to and from I/O devices.
- Memory management statements that manipulate storage.
- Sequence control statements that transfer the flow of control between logical units.
- Condition handling statements that allow a program to intercept and recover from run-time errors.
- Null statements that perform no action but function as placeholders.

All PL/I statements, except the assignment statement, consist of an optional label, followed by a keyword and statement body, and end with a semicolon. Subsequent sections of this manual describe each type of statement in detail. For reference, Appendix B contains a complete alphabetical list of PL/I statement formats.

2.2 Blocks

PL/I is a block-structured language. This means that you group one or more statements into logical units called blocks. A block is a collection of statements in which declared variables are known. Inside a block, you can declare variables and, for certain variables, you can allocate and free storage. You can nest blocks in one another, but not overlap them.

There are two types of blocks: BEGIN blocks and PROCEDURE blocks. A BEGIN block is a sequence of statements delimited by BEGIN and END statements. A PROCEDURE block is delimited by PROCEDURE and END statements.

A BEGIN block has the following format,

```
[label:]
      BEGIN;
      statement-1;
          .
          .
          .
      statement-n;
      END [label];
```

where statement-1 through statement-n are any PL/I statements constituting the body of the block. BEGIN blocks can contain nested PROCEDURE blocks, and nested BEGIN blocks. In PL/I, the label option for the END statement does not automatically cause the block to balance, as it does in some full language implementations.

A PROCEDURE block has the following format,

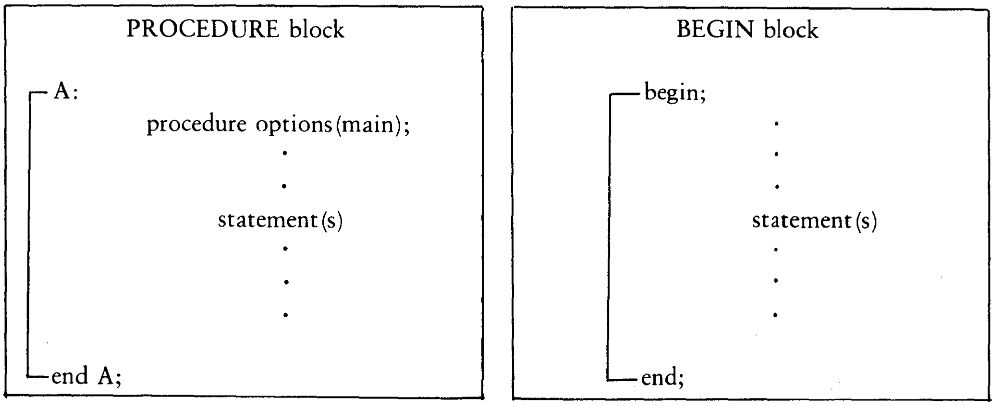
```
proc-name:
            PROCEDURE-statement;
            statement-1;

                    .
                    .
                    .

               statement-n;
               END [proc-name];
```

where proc-name identifies the procedure, and statement-1 through statement-n are any PL/I statements.

Note: the proc-name is optional for the END statement, but if included it must match the proc-name label for the PROCEDURE statement.

The essential difference between a BEGIN block and a PROCEDURE block is how they receive control during program execution. Control flows into a BEGIN block in the usual sequential manner. At this point, the block becomes active. When control transfers, programmatically, outside the block, or its corresponding END statement executes, the block terminates.

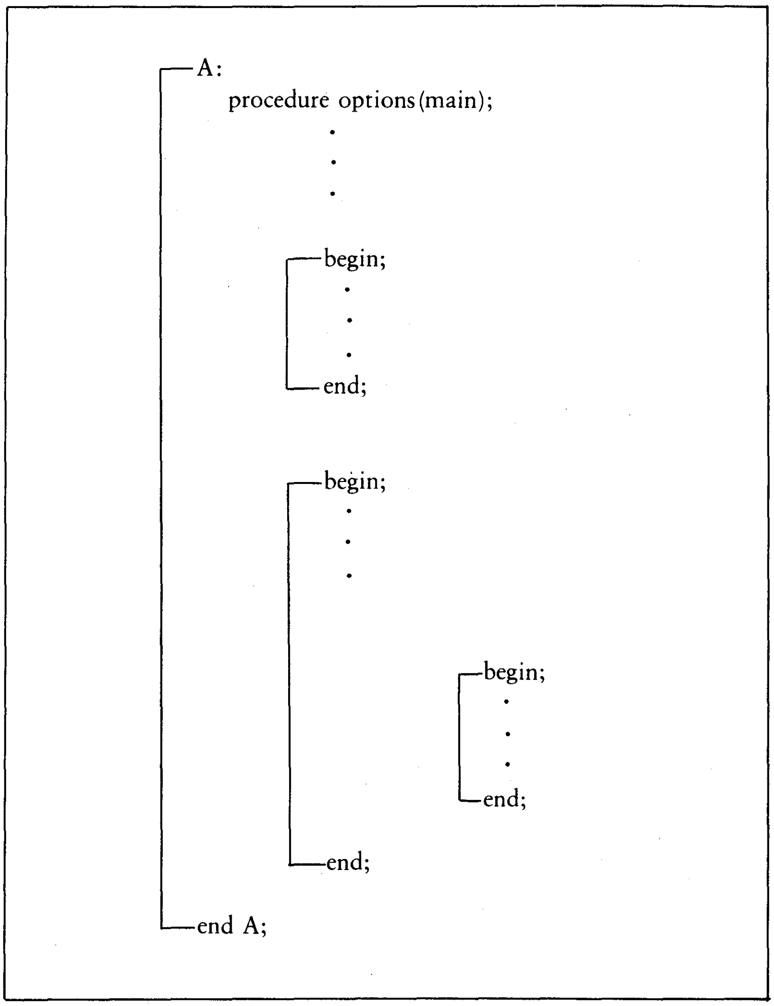
PL/I skips PROCEDURE blocks during the usual execution seauence, and they receive control only when invoked (see Section 2.5.) Figure 2-1 illustrates the block concept.



```
PROCEDURE block

A:
        procedure options(main);
                 .
                 .
           statement(s)
                 .
                 .
                 .
end A;
```

```
BEGIN block

begin;
            .
            .
            .
      statement(s)
            .
            .
            .
end;
```

Figure 2-1. BEGIN and PROCEDURE Blocks



```
A:
   procedure options(main);
         .
         .
         .
      begin;
         .
         .
         .
      end;

      begin;
         .
         .
         .
            begin;
               .
               .
               .
            end;
      end;
end A;
```

Figure 2-1. (continued)

2.3 Internal vs. External Blocks

In PL/I, each block is categorized as either internal or external depending on its relationship with other blocks.

Note: an external procedure is separate from other blocks. The procedure is not contained, nested, in any other block. Thus the main procedure is always an external procedure. An internal procedure is one that is contained in an encompassing block.

A PL/I program can have one or more external procedures that contain nested internal procedures or blocks. Each external procedure can be separately compiled and linked together to form a runnable program. One of the external procedures forming the program must be the main procedure.

In Figure 2-2(a), blocks P1, P2, and P3 are all external but the BEGIN block is internal to P3. In Figure 2-2(b), P1 is the external block, and P2, P3, and the BEGIN block are all internal. The format of the main procedure is:

```
proc-name:
        PROCEDURE OPTIONS(MAIN);
                        .
                        .
                        .
        Statements or Blocks
                        .
                        .
                        .
        END [proc-name];
```

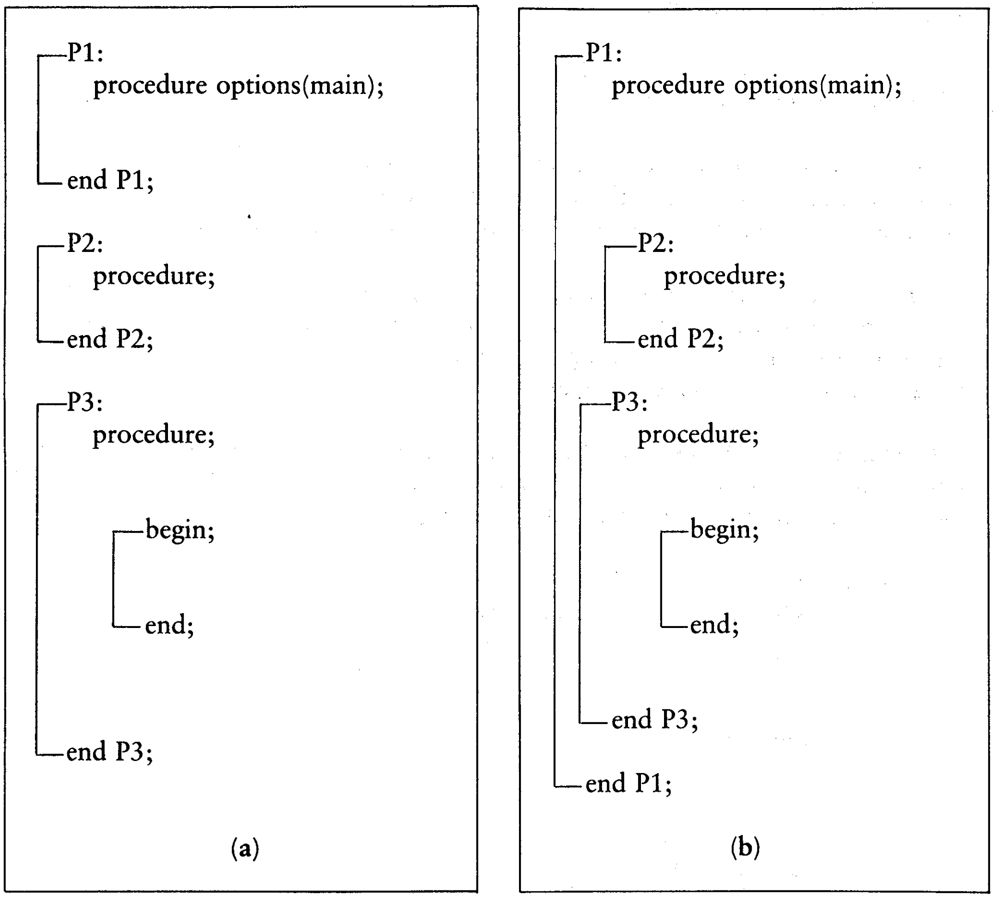


Figure 2-2. Internal and External Blocks

The *PL/I Language Programming Guide* contains specific examples of program structure and how you can separately compile, link, and load external procedures.

2.4 Scope of Variables

The scope of a variable is the set of blocks in which the variable is known. Variables can be either local or external relative to a block in which they appear.

When you declare a variable in a block, you can reference it in that block or any contained block. The variable is said to be local to that block because you cannot reference it outside the block where you declare it. In a contained block, a reference to a variable declared in a containing block is called an up-level reference.

The following example illustrates the concept of scope:

```
P1:
  procedure;
  declare
  (a,b) fixed binary(7);
  a = 2;        /* a is local to P1 */
  b = 3;        /* b is local to P1 */
  P2:
    procedure;
    declare
    b fixed binary(7);
    b = 2;    /* b is local to P2 */
    a = a*b; /* b here refers to P2 b, not P1 b */
  end P2;
  put list (a,b);

end P1;
```

PL/I creates a new variable b in block P2 because it is a declared variable in that block. The PUT LIST statement is outside P2; therefore, the value of the variable b of P1 is 3. Because there is no declaration for the identifier a in P2, a is an up-level reference to the variable a declared in P1, and the assignment statement in P2 changes its value. Thus, this code sequence produces the values 4 and 3.

Any variable declared as EXTERNAL is known to all blocks in which it is declared as EXTERNAL and in all contained blocks unless redeclared without the EXTERNAL attribute. Two declarations of the same variable name denote separate storage locations unless both specify the EXTERNAL attribute.

```
┌─P1:
│     procedure;
│     declare
│     z fixed binary external;
│                  .
│                  .
│                  .
│  ┌─P2:
│  │     procedure;
│  │     declare
│  │     z fixed binary external;
│  │                  .
│  │                  .
│  │                  .
│  │  ┌─P3:
│  │  │  ┌─begin;
│  │  │  │  declare
│  │  │  │  z float binary; /* not external */
│  │  │  │                  .
│  │  │  │                  .
│  │  │  └─end;
│  │  └─end P3;
│  └─end P2;
└─end P1;
```

In this code sequence, the variable z in P1 and P2 refers to the same external variable, but variable z in P3 is a local variable and is distinct from the external variable z.

```
┌─P1:
│   procedure options(main);
│    declare x float binary;
│             .
│             .
│             .
│  ┌─begin;
│  │     declare x fixed;
│  │              .
│  │              .
│  │              .
│  │              .
│  └─end;
│             .
│             .
│             .
│ ┌─P2:
│ │   procedure;
│ │   declare x character(10) varying;
│ │              .
│ │              .
│ │              .
│ └─end P2;
└─end P1;
```

In this sequence, the scope of x is limited to each block in which it is declared. Although the name is identical in each declaration, the Compiler treats each one as a completely different variable with its own data type, and stores them in different memory locations.

2.5 Procedure Blocks

In PL/I, there are two types of procedures: subroutines and functions. Both types perform a specific task and are logically separate from the rest of the program. Both types can execute the same sequence of code one or more times without duplicating the code at each occurrence.

You invoke or call a subroutine and, optionally, pass data items to it in an argument list. The subroutine then manipulates the data and, optionally, returns it to the invoking procedure. Control resumes at the statement immediately following the invocation.

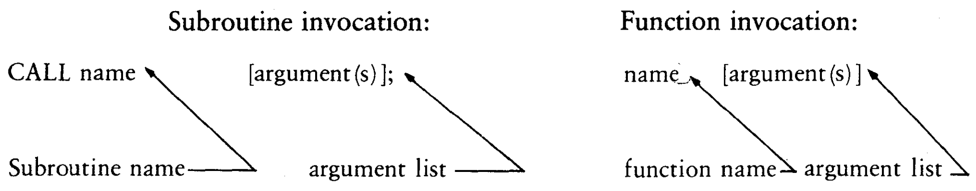
A function is a procedure that manipulates data items and then returns a single value. You invoke a function by referencing its function name and argument list in a statement. Control passes to the function that performs its task and then returns a single value that replaces the function reference. Control then resumes at the point of reference.

2.6 The CALL Statement

The general form of the CALL statement is

```
CALL proc-name[(sub-1,...,sub-n)] [(arg-1,...,arg-m)];
```

where sub-1 through sub-n are optional subscripts that are required only when proc-name is a subscripted entry variable (Section 3.3.2). Arg-1 through arg-m represent the actual parameters passed to the procedure. Figure 2-3 illustrates the invocation of subroutines and functions.



example:

```
call print_header;
call compute(base_pay,overtime);
```

example:

```
point = 3.14/sin(A);
put list (sum(X,Y));
```

Figure 2-3. Subroutine and Function Invocation

2.7 The RETURN Statement

The RETURN statement returns control to the point in the calling block immediately following the procedure invocation. It also returns a value if the procedure is a function procedure.

The general form of the RETURN statement is

RETURN [(return-exp)];

where return-exp is the function value the procedure returns to the calling point. When necessary, PL/I converts the returned value to conform to the attributes specified in the RETURN option of the procedure statement.

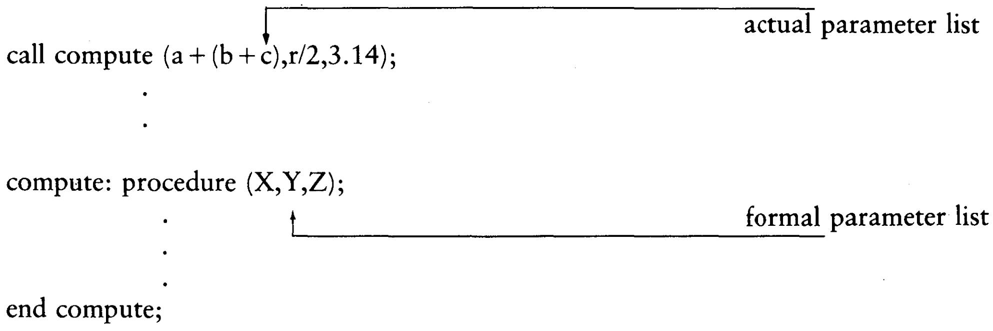
The RETURN statement ends the procedure block that contains it. If the main procedure has the RETURNS attribute, PL/I returns control to the operating system.

Some examples of valid RETURN statements are shown below:

```
return;
return (X**2);
return (F(A,(B)));
```

2.8 Actual and Formal Parameters

The data items that you transmit to a procedure are called the actual parameters, while the data items expected by a procedure and defined in the PROCEDURE statement, are called the formal parameters. Upon invocation of a procedure block, PL/I pairs each actual parameter with its corresponding formal parameter as shown in Figure 2-4 below.



```
call compute (a+(b+c),r/2,3.14);
        .
        .
compute: procedure (X,Y,Z);
        .
        .
        .
end compute;
```

Figure 2-4. Actual and Formal Parameters

When you pass the actual parameter by reference, the actual parameter and corresponding formal parameter share storage. In this case, any changes made to the formal parameter in the invoked procedure change the value of actual parameter of the invoking block.

When you pass the actual parameter by value, the actual and formal parameters do not share storage. In this case, PL/I passes a copy of the actual parameter to the invoked procedure, so that any changes to the formal parameter affect only the copy, not the actual parameter value.

The following example program illustrates parameter passing.

```
┌─A:
│  procedure;
│  declare
│     ACTUAL fixed binary,
│     DUMMY  fixed binary;
│
│  call X(ACTUAL);
│  call X((DUMMY));
│
│ ┌─X:
│ │  procedure (FORMAL);
│ │  declare FORMAL fixed binary;
│ │  FORMAL = 3;
│ └─end X;
│
└─end A;
```

PL/I passes ACTUAL by reference. Therefore, the assignment statement in the procedure X changes the value of ACTUAL throughout the program. PL/I passes DUMMY by value. Thus the procedure only changes a copy of the value inside the procedure.

PL/I passes actual parameters by reference when the data attributes of the actual parameter are the same as the data attributes of the formal parameter. PL/I passes an actual parameter by value when it is one of the following:

- a constant
- an entry name
- an expression consisting of variable references and operators
- a variable reference enclosed in parentheses
- a function invocation
- a variable expression whose data type does not match that of the formal parameter

In the latter case, PL/I converts the actual parameter to the type, precision, and scale of the formal parameter. The following program illustrates this concept:

```
A:
  procedure;
  declare
     X character (7),
     (Y,Z) fixed binary;

  call p(X,(Y),Z);

  p:
       procedure(A,B,C);
       declare
          A character (7),
          B fixed binary,
          C float binary;

       A = 'Digital';
       B = 100;
       C = 2.5E2;

  end p;

end A;
```

The CALL statement sends the procedure three actual parameters X, Y, and Z corresponding to the three formal parameters A, B, and C. PL/I passes the first actual parameter by reference because it matches the formal parameter, and the second parameter by value because it occurs as an expression. PL/I converts the third parameter to the FLOAT data type and passes it by value.

2.9 The PROCEDURE Statement

In PL/I, you can define a procedure with a procedure statement at any point in a program. However, for readability you should place all procedures together in a single section at the end of the main program. The main program is, itself, a single-procedure definition.

The procedure statement identifies the entry point to the procedure, delimits the beginning of the procedure block, defines the formal parameter list, and gives the attributes of the returned value for functions. The procedure can consist of a sequence of one or more statements including the corresponding END statement that ends the procedure definition. The END statement can also be the exit point of the procedure, although embedded RETURN statements can appear within the procedure body.

The general form of the procedure statement is,

```
proc-name: PROCEDURE[(parm-1,..., parm-n)]
           [OPTIONS(option,...)] [RETURNS(attribute-list)]
           [RECURSIVE];
```

where parm-1 through parm-n are the formal parameters for the procedure which you must declare within the procedure body at the principle block level. A formal parameter can be,

- a scalar variable
- an array
- a major structure

but cannot have the attributes:

- STATIC
- AUTOMATIC
- BASED
- EXTERNAL

[OPTIONS(option,...)] defines a list of one or more of the options MAIN, STACK(b), or EXTERNAL.

The MAIN option identifies the procedure as the first procedure to receive control when the program begins execution.

The STACK(b) option sets the size of the run-time stack to the number of bytes specified by b. The default value is 512 bytes.

The EXTERNAL option identifies the procedure as an externally compiled procedure. It is often useful to group separately compiled procedures into a single compilation, where the procedures reference the same global data. According to the Subset G standard, you must compile each subroutine separately, and duplicate the global data area in each compilation. You can then combine the individual modules using the linkage editor to produce the object module.

In PL/I, an EXTERNAL option in a procedure heading makes the procedure accessible outside the module.

Note: for compatibility with future implementations of PL/I from Digital Research, you should only mark the top level procedures as OPTIONS(EXTERNAL), and you should declare all globally accessed data as STATIC. A compilation containing a group of EXTERNAL procedures should consist of subroutines only, with no main program.

The following code sequence shows an example of how to use the EXTERNAL option:

```
module:
    procedure;
    declare
       1 global_data static,
         2 a_field character(20) varying initial(''),
         2 b_field fixed initial(0),
         2 c_field float initial(0);
    set_a:
      procedure (c) options(external);
      declare c character(20) varying;
      a_field = c;
    end set_a;
    set_b:
      procedure (x) options(external);
      declare x fixed;
      b_field = x;
    end set_b;
    set_c:
      procedure (y) options(external);
      declare y float;
      c_field = y;
    end set_c;
    sum:
      procedure returns(float) options(external);
      return (b_field + c_field);
    end sum;
    display:
      procedure options(external);
      put skip list(a_field,b_field,c_field);
    end display;
end module;
```

This code defines five external procedures: set_a, set_b, set_c, sum, and display. These procedures are then accessed in the code sequence shown below:

```
call_ext:
    procedure options(main);
    declare
        set_a entry (character(20) varying),
        set_b entry (fixed),
        set_c entry (float),
        sum returns(float),
        display entry;
    call set_a('Johnson,J');
    call set_b(25);
    call set_c(5.50);
    put skip list(sum());
    call display();
end call_ext;
```

These two code sequences when compiled separately and linked together, form a single runable program.

PL/I requires the RETURNS attribute list for a function procedure to give the characteristics of the value returned by the function.

The RECURSIVE attribute indicates that the procedure can activate itself, either directly or indirectly, while the procedure is executing.

2.10 Low-level Organization

The low-level organization of PL/I source text includes a specification of the character set and the rules for forming identifiers, both keywords and declared names, operators, constants, delimiters, and comments.

PL/I is a free-format language. The source program consists of a sequence of ASCII characters that make up lines delimited by carriage return characters. You can enter the source text without regard for column position or specific line format. However, the source text is easier to read and comprehend if you follow some basic formatting rules:

- Place only one statement on a line.
- Use indentation to show the nesting level of blocks and DO-groups.

You can create the PL/I source program using ED the CP/M Context Editor or a similar text editor.

Note: all PL/I source programs must have the filetype PLI.

2.11 The Character Set

The PL/I character set consists of both upper- and lower-case letters, numeric digits, and other symbols. Table 2-1 shows the symbols recognized by PL/I and briefly describes their use.

Table 2-1. PL/I Symbols

Symbol	*Meaning*
=	equal sign (assignment)
+	plus sign (addition)
—	minus sign (subtraction)
*	asterisk (multiplication)
/	slash (division)
(	left parenthesis (delimiter)
)	right parenthesis (delimiter)
,	comma (separator)
.	period (name qualifier)
%	percent symbol (INCLUDE or REPLACE prefix)
'	apostrophe (string delimiter)
;	semicolon (statement terminator)
:	colon (separator for ENTRY or LABEL constant)
^	circumflex (logical Not symbol)
~	tilde (alternative Not symbol)
&	ampersand (logical And symbol)
!	exclamation mark (alternative Or symbol)
\	backslash (alternative Or symbol)
|	vertical bar (logical Or symbol)
>	right angle bracket (greater than)
<	left angle bracket (less than)
_	break or underscore (for readablity in identifiers)
$	dollar sign (valid character in identifiers)
?	question mark (valid character in identifiers)

2.12 Identifiers

An identifier is a string of from one to thirty-one characters that are either letters, digits, or the underscore. The first character must be a letter. PL/I always represents letters internally in upper-case. Therefore, two identifiers that differ only in case represent the same identifier.

PL/I allows the question mark character to be embedded in identifiers to allow access to external system entry points.

Note: it is good practice to avoid embedded question marks to maintain upward compatibility with full PL/I.

Every identifier in the source text of a PL/I program must be either a keyword or a declared name. Keywords are those identifiers that have a special meaning in PL/I when used in a specific context. Examples of keywords are the names of built-in functions, statements, and data attributes. The *PL/I Language Command Summary* contains a complete list of keywords.

Declared names are identifiers whose use or meaning you define in a DECLARE statement (Section 3.6). A keyword can appear in a declaration as a user-defined identifier. The meaning of the identifier depends on how and where it appears. PL/I determines the meaning in context. For example, INDEX is a keyword because it is the name of a PL/I built-in function. However, in the context of the declaration,

```
declare index fixed binary;
```

index is a declared name and not a keyword.

2.13 Constants

Constants are text items that have a fixed literal meaning that cannot change during program execution. In PL/I, the basic constants are:

- arithmetic (Example: 3674.799)
- character string (Example: 'Ada Lovelace')
- bit string (Example: '00010110')

2.14 Delimiters and Separators

Separate items, such as identifiers, must be distinguishable. PL/I recognizes certain characters as delimiters and separators.

Generally, delimiters enclose one or more text items while separators mark the end of one item and the beginning of another. In PL/I, each identifier and arithmetic constant must be preceded and followed by one or more delimiters or separators. Delimiters can be either spaces, operators, or certain special characters.

2.14.1 Spaces

In PL/I, a space can be either a blank, or a tab (CTRL-I). PL/I ignores any carriage return, line-feed, or carriage return line-feed sequence that is embedded in a string constant. For example, the assignment statement,

```
string = 'WHEN YOU HAVE A VERY LONG STRING LIKE THIS, PL/I ALLOWS
YOU TO PUT SOME OF IT ON ANOTHER LINE';
```

assigns the specified character string to the variable string. Any blanks or tabs that precede ALLOWS are included in the string.

2.14.2 Operators

An operator is a symbol for a mathematical or logical operation. There are four types of operators in PL/I as shown in Table 2-2.

Note: operators that consist of two characters, such as >=, are called composite operators and must not be separated by blanks or other spaces.

Table 2-2. PL/I Operators

Symbol	*Meaning*
	Arithmetic Operators
+	addition or prefix plus
−	subtraction or prefix minus
*	multiplication
/	division
**	exponentiation

Table 2-2. (continued)

Symbol	*Meaning*
	Comparison Operators
>	greater than
^> or ~>	not greater than
>=	greater than or equal to
=	equal to
^= or ~=	not equal to
<=	less than or equal to
<	less than
^< or ~<	not less than
	Bit-string Operators
^ or ~	not
&	and
! or |	or
	The String Operator
!! or ||	concatenate

2.14.3 Special Characters

Table 2-3 shows the special characters that can also function as delimiters or separators in PL/I. Subsequent sections of the manual contain examples of their use.

Table 2-3. Special Character Delimiters and Separators

Character	*Function*
:	A colon separates ENTRY and LABEL constants.
;	A semicolon terminates statements.
,	A comma separates elements of a list.

Table 2-3. (continued)

Character	*Function*
.	A period separates items in a qualified name.
'	A single quote is a delimiter for the specification of character and bit-string constants.
->	The arrow is a composite pair of the minus sign and the right angle bracket. It is a separator in a pointer qualified reference.
=	An equal sign is a separator in an assignment statement.
(	Left parenthesis.
)	A right parenthesis together with a left parenthesis is used as a delimiter pair to enclose lists and extents, define the order of evaluation of expressions, and separate keywords from statements and option names.

2.14.4 Comments

Comments provide documentary text in a PL/I source program. The Compiler ignores comments, so you can place them wherever a delimiter is appropriate. Precede a comment by the composite pair /* and end the comment by the reverse composite pair */. For example:

```
        .
        .
        .
get list(name);   /* read the name */
        .
        .
        .
```

2.15 Preprocessor Statements

PL/I allows modification of the source program or inclusion of external source files at compile time through the use of preprocessor statements. Preprocessor statements are identified by a leading % symbol before the keyword:

INCLUDE or REPLACE

2.15.1 The %INCLUDE Statement

The %INCLUDE statement copies PL/I source text from an external file at compile time. The statement is useful for filling in a structure declaration or format list. The form of the statement is

%INCLUDE 'filespec';

where filespec designates the file to copy into the source program. Filespec must be a standard CP/M file specification, [d:]filename[.typ], and must be enclosed in parentheses. If there is no drive specification, PL/I assumes the drive containing the source program. When the Compiler encounters the %INCLUDE statement in the source file, it begins reading the file specified by %INCLUDE. When the Compiler reaches the end of the %INCLUDE file, it resumes reading the original source file.

The following code sequence is an example of the %INCLUDE statement:

```
f:
  procedure;
  declare a fixed binary;
  %include 'struc.lib';
  declare c float;
     .
     .
     .
end f;
```

The Compiler includes the source text from the file struc.lib at the point of the %INCLUDE statement.

Note: PL/I does not allow nested %INCLUDE statements.

2.15.2 The %REPLACE Statement

The %REPLACE statement allows the Compiler to replace constants for defined identifiers throughout the source program. The form of the statement is:

%REPLACE identifier BY constant;

The Compiler replaces every occurrence of the given identifier in the source text with the specified constant. The constant can be any string or unsigned arithmetic constant. You can write multiple %REPLACE statements as a single %REPLACE statement, with the elements separated by commas.

For example, the statement

```
%replace true by '1'b;
```

replaces all occurrences of true by the constant bit string'1'b, so that the Compiler interprets the statement,

```
do while (true);
```

as:

```
do while ('1'b);
```

PL/I requires that all %REPLACE statements occur at the outer block level before any nested inner blocks.

Note: to facilitate program maintenance and debugging, you should write all %REPLACE statements directly following the procedure heading.

End of Section 2

Section 3 Data Types and Attributes

Data items in a PL/I program are either constants or variables. A constant is a data item whose value does not change during program execution, while the value of a variable can change during execution.

Every data item is associated with a set of properties called attributes that include such things as a range of subscript values, the operations that can be applied, and the amount of storage required. The DECLARE statement explicitly assigns attributes to data variables, while in some cases, such as constants, attributes are implicitly assigned by system defaults (see Section 3.6).

Data variables can represent single data items. A single data item, either a variable or constant, is called a scalar. Data variables can also represent multiple data items called aggregates. (Section 5 describes data aggregates.)

PL/I supports six types of data:

- arithmetic
- string
- pointer
- label
- entry
- file

The following sections describe each of these data types in detail.

3.1 Arithmetic Data

PL/I supports three types of arithmetic data:

- FIXED BINARY for representing integer values
- FLOAT BINARY for representing very large or very small numbers, with the decimal point allowed to float
- FIXED DECIMAL for representing decimal integers or fractions with a fixed number of fractional digits

Each arithmetic data item has an associated precision and scale value expressed as integer constants p and q enclosed in parentheses. The precision p specifies the number of decimal or binary digits the data item can contain. For FIXED DECIMAL numbers, the scale q specifies the number of digits to the right of the decimal point. If you do not explicitly declare the precision and scale of a variable in a DECLARE statement, PL/I implicitly supplies them according to default rules.

3.1.1 FIXED BINARY

FIXED BINARY data represents integers. A variable declared as FIXED BINARY[(p)] is an integer that has p binary digits. The maximum range of p is:

$1 <= p <= 15$

PL/I internally represents this data type in two's complement form. Therefore, the range of a FIXED BINARY number is from −32768 to +32767.

The amount of storage PL/I allocates for a FIXED BINARY number depends on the precision you declare.

If $p <= 7$, then PL/I allocates one byte;
if $p > 7$, then PL/I allocates two bytes.

The default precision for FIXED BINARY is fifteen. Declaring a variable as FIXED, BINARY, or FIXED BINARY is equivalent to declaring it as FIXED BINARY(15).

Note: assigning values to FIXED BINARY variables outside the legal range produces undefined results.

PL/I treats decimal integers in the source program as FIXED BINARY data only if they appear in contexts that require FIXED BINARY values, such as subscripts or arithmetic operations involving other FIXED BINARY data. Otherwise, constants default to FIXED DECIMAL. In PL/I, conversion from other types of data usually occurs with truncation (Section 4 has the conversion rules). For example, the following code assigns the value one to the variable I.

```
declare I fixed binary;
I=1.99;
```

3.1.2 FIXED DECIMAL

FIXED DECIMAL data is used for calculations where exact decimal values must be maintained, as for example, in commercial applications. FIXED DECIMAL data with a zero scale factor can also represent integer data.

A variable declared as FIXED DECIMAL[(p[,q])] is a decimal number with a sign, a total of p decimal digits, with q digits to the right of the decimal point. The maximum number of digits p for FIXED DECIMAL is fifteen, and the scale q must be less than the precision. The range of a FIXED DECIMAL number x is

$$-10^{**}(p-q) < |x| < 10^{**}(p-q)$$

where:

$$1 <= p <= 15 \quad \text{and} \quad 0 <= q <= p$$

In PL/I, all decimal constants, except those used in a FIXED BINARY context, with or without a decimal point default to FIXED DECIMAL. The default precision and scale for FIXED DECIMAL is (7,0). Also, the form of a constant implicitly determines its default precision and scale. For example:

3.25 defaults to FIXED DECIMAL(3,2)
302 defaults to FIXED DECIMAL(3,0)

Internally, PL/I represents decimal numbers in nine's complement BCD format. The number of bytes occupied by a FIXED DECIMAL number depends on its declared precison. If the precision is p, the number of bytes reserved is the integer part of,

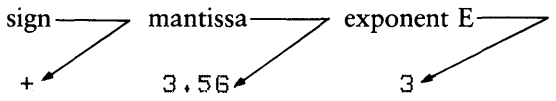
$$(p+2)/2$$

resulting in a minimum of one byte and a maximum of eight bytes.

PL/I truncates any value whose scale is greater than the FIXED DECIMAL variable to which it is assigned. Also, PL/I signals a FIXED OVERFLOW error if a value assigned to the variable has more significant digits to the left of the decimal point than the declared precision of the variable allows.

3.1.3 FLOAT BINARY

FLOAT BINARY data is useful in scientific applications for representing very large or very small numbers. A variable declared as FLOAT BINARY(p) has three parts: a sign, s; p binary digits that are the fraction, or mantissa, and represent significant digits of the number; and an integer exponent e, that represents the scale factor. For example, the FLOAT BINARY number 3.56E3 has the following parts:



PL/I supports both single-precision and double-precision FLOAT BINARY numbers. The following table shows the allowed precisions and the approximate range of magnitudes for each type.

Table 3-1. PL/I FLOAT BINARY Numbers

Type	*Precision p*	*Range r*
single	1 <= p <= 24	5.88*10** −39 <= |x| <= 3.40*10**38
double	25 <= p <= 53	9.46*10** −308 <= |x| <= 1.80*10**308

The default precision for FLOAT BINARY is twenty-four, so declaring a variable FLOAT is equivalent to declaring it FLOAT BINARY(24).

A FLOAT BINARY constant is a number expressed in scientific notation as a sequence of decimal digits with an optional decimal point followed by the letter E, followed by an optionally signed decimal integer exponent. For example, the following code

```
A = 2.3E2;
B = -4.67E+5;
C = 1.98E-2;
```

assigns the value 230 to A, -467000 to B, and 0.0198 to C.

You can mix constants of different data types in an expression. PL/I automatically converts to the common data type before evaluating the expression. For example, in the following code sequence,

```
declare p float binary;
        .
        .
        .
p = p + 3.14159;
```

PL/I converts the FIXED DECIMAL constant 3.14159 to FLOAT BINARY format before performing the addition.

3.2 String Data

PL/I supports two types of string data:

- character string
- bit string

A character string is any sequence of ASCII characters, including the empty or null sequence. A bit string is a sequence of bits. The length of a string is the number of characters or bits in the string. The following sections describe each type of string data.

3.2.1 Character-string Data

A variable declared as CHARACTER(n) is a character string of length n, where n is a value between 1 and 254. For example, the statement

```
declare A character(10);
```

defines the variable A as a character string ten characters long. If a character string assigned to A is shorter than A, PL/I pads the string with blanks on the right to the length of A. If a longer string is assigned to A, PL/I truncates the string on the right.

Character-string constants are a sequence of characters enclosed in apostrophes. If an apostrophe is part of the string, it is written as two consecutive apostrophes. Thus, the string constant whose value is,

What's Happening?

is written as:

'What''s Happening?'

The null or empty character string has a length of zero and is defined by using two consecutive apostrophes.

Character-string variables can also have the VARYING attribute indicating that the variable can represent varying length strings to a maximum length of n. For example, the statement

```
declare A character(10) varying;
```

defines A to represent any character-string value whose length is not greater than ten.

PL/I allows control characters in string constants. The circumflex character ^ in a string constant indicates a control character. PL/I masks the high-order three bits of the character to zero, thus converting the string ^M, or ^m, to a carriage return character. Similarly, it converts the string ^I to the horizontal tab character. PL/I translates a double circumflex ^^ within the string to a single ^ character.

Note: PL/I programs should avoid using the control character feature if compatibility is a requirement, because the circumflex is not available in other PL/I implementations.

3.2.2 Bit-string Data

Bit strings represent logical data items. A bit string containing all zero-bits is false; a bit string containing any one-bits is true.

A variable declared as BIT(n) is a bit-string data item containing n binary digits, where n is a value between one and sixteen. For example, the statement,

```
declare A bit(3);
```

defines a bit string of length three. Bit-string assignments follow the same rules as for character strings, except that padding is done with zero-bits instead of blanks.

Note: bit-string variables cannot have the VARYING attribute.

You can write bit-string constants in any of four different formats. Each format corresponds to a base which is the number of bits used to represent each digit in the constant. A bit-string constant is a sequence of digits and letters enclosed in apostrophes followed by the letter B, and optionally followed by a digit indicating the base. The default base, B or B1, is two digits. The following table shows the various formats.

Table 3-2. Bit-String Constant Formats

Format	*Base*	*Digits and/or Characters in Representation*
B	2	0,1
B1	2	0,1
B2	4	0,1,2,3
B3	8	0,1,2,3,4,5,6,7
B4	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Note: the characters and/or digits used in the sequence must be valid for the base specified by the format.

The following examples illustrate the equivalence of the optional formats to the base 2 format:

```
'101'B1  is  equivalent  to  '101'B
'101'B2  is  equivalent  to  '010001'B
'101'B3  is  equivalent  to  '001000001'B
'101'B4  is  equivalent  to  '000100000001'B
'9A'B4   is  equivalent  to  '10011010'B
'77'B3   is  equivalent  to  '111111'B
```

3.3 Control Data Items

Control data items control the flow of program execution. Statement labels and procedure names are examples of control data constants.

3.3.1 LABEL Data

LABEL data consists of label constants and label variables. A label constant is a label identifier that prefixes an executable statement. A label variable is a variable defined in a DECLARE statement with the LABEL attribute.

Assignments of label constants or other label variables can be made to a label variable following the same rules as assignments of other types of variables.

Both label constants and label variables are subject to the same scope rules as declared names. A LABEL data item is known only within the block in which it is declared explicitly by a DECLARE statement, or implicitly by its use as a label constant.

Label constants can be subscripted by a single, optionally signed, integer constant. All occurrences of subscripted labels with the same identifier in a single block constitute an implicit declaration of a constant-label array for that block. The occurrence of the same label name within any other block, including a contained block, defines a new declaration local to that block. Any such implicitly defined constant-label array is defined only for those subscripts that occur in its corresponding block. You can explicitly define label variables to be singly subscripted arrays in a DECLARE statement.

Note: in PL/I, the only operators that you can use with LABEL data are the equal (=) and not equal (^= or ~=) comparison operators.

3.3.2 ENTRY Data

In PL/I all ENTRY data items are either entry constants or entry variables. Entry constants correspond to either internal procedures, or to separately compiled external procedures. Entry variables are data items that can take on entry-constant values during program execution.

The calling program must use an ENTRY declaration to define the characteristics of the formal parameters and returned values for all externally compiled procedures.

Note: you must ensure that the ENTRY declaration matches the externally defined procedure, so the linkage editor can properly combine the program segments.

Variables that take on entry constant values are also defined with an ENTRY declaration. If required by the application, entry variables can be subscripted, but entry constants cannot. As with LABEL data, the only operators used with ENTRY data are the equal and not equal comparison operators.

The ENTRY attribute defines an identifier as an ENTRY data item, giving the attributes for the formal parameters, and the optional returned value attributes if the entry item is a function.

The general form for an ENTRY statement is

```
proc-name [(bound-pair-1,...,bound-pair-n)] [VARIABLE]
          [ENTRY [(att-1,...,att-m)]
          [RETURNS(return-att)];
```

where the attributes can be in any order, but must specify either ENTRY or RETURNS.

The identifiers are given in the following manner:

- proc-name gives the entry-data item name
- bound-pair-1,...,bound-pair-n gives the optional bound-pair list
- att-1,...,att-m gives the list of formal parameter attributes
- return-att gives return-value attribute for a function entry item

The VARIABLE attribute indicates that the data item is an entry variable that must be assigned an entry constant value during program execution. The bound-pair list is only valid if the item has the VARIABLE attribute. You can omit the list of formal parameter attributes if the procedure does not require any parameters. In this case, you can also omit the ENTRY attribute if you specify the RETURNS attribute.

If a particular parameter has the dimension attribute, it must appear as the first attribute. If the parameter is a structure, the structuring information that the level numbers provide must precede the attribute definition. PL/I does not permit attribute factoring in the list att-1 through att-m.

Some examples of valid ENTRY statements are given below:

```
declare X entry;
declare Y entry variable;
declare P (0:10) entry(fixed,float) variable;
declare Q entry(1, 2 fixed, 2 float,(5:10) decimal);
declare R returns (character(10));
```

The following code sequence illustrates entry data items:

```
declare
   (X,Y) float binary,
   A entry variable,
   F(3) entry(float) returns(float) variable,
   ZZ entry(float) returns(float);
P1:
  procedure;
  X=5;
  end P1;
P2:
  procedure;
  X=25;
  end P2;
Y=9;
if Y = 5 then
   A = P1;
else
   A = P2;
call A;
F(2) = ZZ;
Y = F(2)(X);
put list(Y);
```

3.4 POINTER Data

POINTER data addresses specific locations in memory. The value of a POINTER data item is the address of a variable in the program. The general form of a POINTER variable declaration is:

DECLARE X POINTER;

PL/I does not define any conversion between POINTER and other data types, so an assignment statement can only assign pointer variables to other pointer variables. Also, pointer variables cannot be output to a STREAM file. As with LABEL and ENTRY data, the only operators defined for POINTER data are the equal and not equal comparison operators. Two pointers are equal if they represent identical storage locations.

You can use POINTER data with based variables to dynamically manage storage. Section 7.2 describes based variables.

3.5 FILE Data

In PL/I, FILE data items consist of file constants and file variables that access external data. A file constant declaration takes the general form:

```
DECLARE file_id FILE;
```

A file variable declaration takes the general form,

```
DECLARE file_id FILE VARIABLE;
```

where file_id is a PL/I identifier assigned to represent the file. If file_id is not a parameter, PL/I automatically treats the identifier as EXTERNAL, so that it accesses the same data set in all modules that declare it EXTERNAL.

If you do not open the file explicitly with an OPEN statement including the TITLE option, PL/I accesses the disk file file_id.DAT on the default drive.

Section 10 presents FILE data in more detail. The *PL/I Language Programming Guide* contains examples of FILE data use.

3.6 The DECLARE Statement

In PL/I, you must use the DECLARE statement to define all variable names in a program that are not the names of built-in functions or pseudo-variables (Section 6.8). File constants and variables must also be defined in a DECLARE statement. Control constants, such as statement labels and procedure names, are declared implicitly by their use in a program.

The DECLARE statement associates each variable name with the proper attributes for the declared data type. The general form of the DECLARE statement for scalar variables is

```
DECLARE name [attribute-list];
```

where name is the variable identifier, and attribute list is one or more characteristics of the variable name. Multiple attributes can appear in any order but must be separated by spaces.

The following examples illustrate valid PL/I DECLARE statements:

```
declare x fixed binary;
declare pi float binary(53);
declare overtime_pay fixed decimal(5,2) initial(000.00);
declare (first_name,last_name) character(20) varying;
declare EOF bit(1) initial('1'b);
declare list_head pointer static initial(null);
```

3.7 Multiple Declarations

For convenience and simplicity, PL/I allows multiple declarations in a single statement. In general, you can write any sequence of DECLARE statements of the form,

DECLARE definition-1;
DECLARE definition-2;
.
.
.
DECLARE definition-n;

in the equivalent form,

DECLARE definition-1, definition-2, ... definition-n;

where each definition item is separated by commas and zero or more spaces, and the DECLARE statement is terminated by a semicolon.

If several item definitions share the same attributes, you can factor them to the right. That is, you can write a sequence of definitions of the form,

item-1 attr-A, item-2 attr-A, ... item-n attr-A

in an equivalent factored form:

(item-1, item-2, ... item-n) attr-A

Repeated applications of this rule are also allowed. For example, the statement

```
declare ((A,B) fixed binary, C float binary) static external;
```

is equivalent to:

```
declare A fixed binary static external,
        B fixed binary static external,
        C float binary static external;
```

The ordering of attributes is unimportant, with the exception that the dimension list attribute for an array must follow the array name and precede other attributes. Also, the level numbers for members of structures must precede the member name. Both attributes can be factored.

Because a dimension list is on the right of the name to which it applies, it is factored to the right as above. However, because level numbers precede their member names, they are factored to the left. A sequence of the form,

level-k item-1, level-k item-2, ... level-k item-n

is equivalent to the sequence:

level-k (item-1, item-2, ... item-n)

For example, the statement

```
declare 1 A based,
          2 (B fixed binary,
             C character(2));
```

is equivalent to:

```
declare 1 A based,
          2 B fixed binary,
          2 C character(2);
```

3.8 Default Attributes

An attribute list cannot contain conflicting attributes, such as two data types, or two storage class attributes. If you do not specify a complete set of attributes in a DECLARE statement, then the Compiler supplies the attributes according to the following default rules:

- If no attribute is specified, FIXED BINARY(15) is assumed.
- If DECIMAL or BINARY is specified without FIXED or FLOAT, then FIXED is assumed.
- If FIXED or FLOAT is specified without BINARY or DECIMAL, then BINARY is assumed.
- If no precision for FIXED BINARY is specified, FIXED BINARY(15) is assumed.
- If no precision and scale for FIXED DECIMAL is specified, FIXED DECIMAL(7,0) is assumed.
- If no precision for FLOAT BINARY is specified, then FLOAT BINARY(24) is assumed.
- If no length is specified for BIT, then BIT(1) is assumed.
- If no length is specified for CHARACTER, then CHARACTER(1) is assumed.

End of Section 3

Section 4
Data Conversion

Data conversion is a process that changes the representation of a given value from one type to another. In PL/I, all conversion involves a source, a target, and a result. The source is the data item being converted; the target is the type to which the source item is being converted, and the result is the actual converted value with the data type of the target.

PL/I performs conversions in the following general categories:

- arithmetic to arithmetic (type and precision)
- arithmetic to string
- string to arithmetic
- format specified in EDIT-directed I/O (see Section 13)

PL/I does not perform conversion for LABEL, ENTRY, POINTER, or FILE data types.

Part of the versatility and power of PL/I lies in your freedom to declare data in a wide variety of types. With this freedom comes a responsibility to understand how the language converts data from one type to another, either explicitly or implicitly.

The following list shows some of the contexts in which PL/I performs default data conversion.

- In an assignment statement, PL/I converts the expression to the type of the variable to which it is assigned.

  ```
  variable = expression;
  ```

- In a RETURN statement, PL/I converts the specified value to the type specified in the RETURNS option of the PROCEDURE statement.

```
proc-name:
          PROCEDURE RETURNS(return-att);
          .
          .
          .
          RETURN (return-exp);
END [proc-name];
```

- In any arithmetic expression, if the operands are not the same type, PL/I converts them to a common type before performing the operation.

```
A + B
A - B
A * B
A / B
A ** B
```

- During I/O processing, PL/I converts to and from character string data when using the PUT or GET statement respectively.

```
PUT LIST (output-list);
GET LIST (input-list);
```

- PL/I converts values specified in some statements to integer values.

```
DO (control-var)...
  .
  .
  .
END;
```

- PL/I has built-in functions (BIFs) that perform specific conversions.

4.1 Arithmetic Conversions

Arithmetic conversions occur in several contexts:

- When an assignment statement assigns an arithmetic expression to an arithmetic variable, PL/I converts the expression to the precision and scale of the target variable.

- When an arithmetic-valued function returns an arithmetic expression, PL/I converts the expression to the data type and precision specified in the RETURNS attribute of the function entry point.

- When an arithmetic infix operator has operands with different data types, PL/I first converts them to a common type as follows:

  - If one operand is FIXED BINARY and the other is FLOAT BINARY, the common type is FLOAT BINARY.

  - If one operand is FIXED BINARY and the other operand is FIXED DECIMAL, the the common type is FIXED DECIMAL.

  - PL/I converts FIXED BINARY(p) to FLOAT BINARY(p).

  - PL/I converts FIXED DECIMAL(p,q) to FLOAT BINARY(p'), where p' = MIN(CEIL(p/3.32),53). MIN and CEIL are PL/I BIFs (Section 15).

After performing the conversions to the common type, PL/I derives the target result as follows:

- If the operands are FLOAT BINARY, then the result is FLOAT BINARY The precision of the result becomes the precision of the greater of the two operands.

- If the operands are FIXED DECIMAL, assume the first operand has precision and scale (p,q), and the second operand has precision and scale (r,s). PL/I derives the precision and scale of the result (p',q') as follows

  If the operation is addition or subtraction, then:

  p' = MIN(15,MAX(p-q,r-s) + MAX(q,s)+1)

  q' = MAX(q,s)

  If the operation is multiplication then:

  p' = MIN(15,p+r+1)

  q' = (q+s)

If the operation is division, then:

$p' = 15$

$q' = 15-(p+q-s)$

Note: exercise caution when dividing FIXED DECIMAL values. The precision and scale of the operands must be such that the divide operation does not produce a negative scale factor. You can use the DIVIDE BIF to control the precision of the quotient.

- If the operands are FIXED BINARY, assume (p) is the precision of the first operand, and (r) is the precision of the second operand. PL/I derives the precision of the result (p') as follows:

  If the operation is addition or subtraction, then:

  $p' = (MIN(15,MAX(p,r)+1))$

  If the operation is multiplication, then:

  $p' = (MIN(15,p+r+1))$

  If the operation is division, then you must use the DIVIDE BIF with a scale factor of zero to produce an integral FIXED BINARY result. Do this for compatibility with the full language.

- In the case of exponentiation, expressed as X**Y, assume that Y is a decimal integer constant.

  If X is FIXED BINARY with precision p and $((p+1)*Y-1) <= 15$, then the result is FIXED BINARY with precision:

  $((p+1)*Y-1)$

  If X is FIXED DECIMAL with precision and scale (p,q) and $((p+1)*Y-1) <= 15$, then the result is FIXED DECIMAL with precision and scale (p',q'):

  $p' = (p+1)*Y-1$

  $q' = q*Y$

In all other cases, PL/I converts the operands to FLOAT BINARY and the result is FLOAT BINARY with the precision being the maximum of the precisions of the converted operands.

- PL/I truncates the result if the precision is insufficient to hold the number. Truncation occurs on the right for FLOAT BINARY data items. In FIXED DECIMAL computations, fractional digits are lost; in FIXED BINARY computations, digits are lost in the most significant portion.

4.2 Arithmetic Conversion Functions

PL/I provides a number of BIFs to control the conversions that take place during expression evaluation. The following sections detail these functions.

4.2.1 The FIXED BIF

The form of the FIXED BIF is

```
FIXED (x,[p[,q]])
```

where X is the variable or expression to be converted to a FIXED arithmetic data type, and p and q specify the target precision and scale. If X is FIXED BINARY, you must specify q = 0. A non-zero scale is valid only if X is FIXED DECIMAL.

If X is FIXED DECIMAL, the result is FIXED DECIMAL. Otherwise, the result is FIXED BINARY.

If p or q is not specified, then the result depends on the precision and scale of X as follows:

X FIXED BINARY (r)	yields FIXED BINARY (r)
X FLOAT BINARY(r)	yields FIXED BINARY(MIN(15,r))
X FIXED DECIMAL(r,s)	yields FIXED DECIMAL(r,s)

4.2.2 The FLOAT BIF

The form of the FLOAT BIF is

FLOAT(x,[p])

where X is the variable or expression to be converted to a FLOAT arithmetic data type, and p is the target precision. If p is not specified, then the result is as follows:

X FIXED BINARY(r)	yields FLOAT BINARY(r)
X FLOAT BINARY(r)	yields FLOAT BINARY(r)
X FIXED DECIMAL(r,s)	yields FLOAT BINARY (MIN(CEIL((r-s)*3.32),53))

4.2.3 The BINARY BIF

The form of the BINARY BIF is

BINARY (X[,p])

where X is the variable or expression to be converted to a BINARY arithmetic data type, and p is the target precision.

If X is FIXED BINARY or FIXED DECIMAL, the result is FIXED BINARY. If X is FLOAT BINARY, then the result is FLOAT BINARY.

If p is not specified, then the result is as follows:

X FLOAT BINARY(r)	yields FLOAT BINARY(r)
X FIXED BINARY(r)	yields FIXED BINARY(r)
X FIXED DECIMAL(r,s)	yields FIXED BINARY (MIN(CEIL((r-s)*3.32)+1,15))

4.2.4 The DECIMAL BIF

The general form of the DECIMAL BIF is

DECIMAL(X[,p[,q]])

where X is the variable or expression to be converted to a FIXED DECIMAL arithmetic data type, and p and q are the precision and scale of the target result. A non-zero scale is valid only if X is FIXED DECIMAL. If p and q are not specified, then the result is as follows:

X FIXED BINARY(r)	yields FIXED DECIMAL(CEIL(r/3.32) + 1,0)
X FLOAT BINARY(r)	yields FIXED DECIMAL(MIN(CEIL(r/3.32),15),0)
X FIXED DECIMAL(r,s)	yields FIXED DECIMAL(r,s)

4.2.5 The DIVIDE BIF

The DIVIDE BIF controls the precision of results for divide operations. The general form is

```
DIVIDE(X,Y,p[,q])
```

where X and Y are any arithmetic expressions, and X is to be divided by Y. p is a FIXED BINARY expression indicating the desired precision, and q is a FIXED BINARY expression indicating the desired scale. If not included, q is assumed to be zero. A non-zero scale is valid only if X and Y are FIXED DECIMAL.

PL/I requires the DIVIDE function for FIXED BINARY division because in the full language, a nonzero scale factor results from such an operation.

4.3 String Conversions

PL/I performs conversions between arithmetic and string data items when they are combined in expressions. The following sections describe the various conversion rules for string operands.

4.3.1 Arithmetic to Bit-string Conversion

When converting from an arithmetic source data type X to a bit-string target, PL/I first converts ABS(X) to FIXED BINARY(p) according to the arithmetic conversion rules. It then converts the FIXED BINARY intermediate value to a bit string of length p.

If the target length is longer than p, PL/I pads the intermediate result on the right with zero-bits. If the target length is less than p, it truncates the right excess bits of the intermediate result.

4.3.2 Arithmetic to Character Conversion

When converting arithmetic data to character data, PL/I first converts the various arithmetic data types to intermediate character strings as follows:

- DECIMAL(p,q), q = 0

  The resulting character string is length p + 3. The characters are composed of the digits of the source, without leading zeros, preceded by a minus sign if the source value is negative, and padded on the left with blanks to produce a character string of length p + 3.

  For example, converting a FIXED DECIMAL(3) data item with value 330 results in the character string b̸b̸b̸330, where b̸ denotes a blank position. Converting the value zero produces five blanks and a single zero digit result.

- DECIMAL(p,q), q > 0

  The resulting character string is also of length p + 3, with the same string format as above, except that the decimal point and the fractional digits are included.

  For example, converting a FIXED DECIMAL(5,2) data item with value -13.25 results in the character string b̸b̸-13.25. PL/I omits leading zeros except for the one immediately preceding the decimal point.

- FIXED BINARY(p)

  PL/I converts the source to FIXED DECIMAL(p'), where p' = CEIL(b̸/3.32) + 1, and then converts the FIXED DECIMAL(p') result to a character string of length p' + 3 with the format described above.

  For example, converting a FIXED BINARY(15) data item with value -32 results in the character string b̸b̸b̸b̸b̸b̸-32.

- FLOAT BINARY(p)

  PL/I converts the fractional part to a FIXED DECIMAL(p'), where p'=CEIL (p/3.32). The resulting character string is of length p'+6 for single precision, or p'+7 for double precision in scientific notation format. That is, the first character is a minus sign if the source value is negative, otherwise the position contains a space. The next position contains the most significant digit of the value, followed by a decimal point, and the remaining p-1 fractional digits. The exponent indicator E follows, with an exponent sign and an exponent value. Single precision exponents have two digits, and double precision exponents have three digits.

For example, converting a FLOAT BINARY(24) data item with value 250.1E1 results in the character string ƀƀƀƀ2.501E+03.

After performing the intermediate conversions, PL/I pads the string on the right with blanks if the target length is greater than the length of the intermediate result. Conversely, if the target length is shorter than the intermediate result, PL/I truncates the string on the right to produce the shorter length.

4.3.3 Bit-string to Arithmetic Conversion

When converting a bit string of length n, where $0 < n <= 15$, to an arithmetic data type, PL/I first converts the string to its FIXED BINARY(15) equivalent. It then converts the FIXED BINARY to the target value according to the rules discussed above.

For example,'1011'B converted to FIXED BINARY(15) yields the value eleven.

4.3.4 Bit to Character-string Conversion

When converting a bit string of length n to a character string of length n, PL/I converts a zero bit to a character zero, and a one bit to a character one. If the target length is longer than the source, PL/I pads the target on the right with blanks. If the target length is shorter than the source length, it truncates the excess right characters.

4.3.5 Character to Arithmetic Conversion

When performing character to arithmetic conversion, the character string source must contain a valid arithmetic constant value. If X is a character string, the built-in arithmetic conversion functions affect any conversions applied to X as follows:

- FIXED(x[,p[q]]) or DECIMAL(x[,p[q]]) returns a FIXED DECIMAL value. If p is not given, then fifteen is assumed.

- BINARY(x[,p]) produces a FIXED BINARY value. If p is not specified, it defaults to fifteen. The result is only the integer portion of X.

- FLOAT(x[,p]) produces a FLOAT BINARY value. If p is not given, p = 53 is assumed. If X is null or contains all blanks, the converted value is zero. If there is insufficient precision to hold the converted value, the run-time system signals OVERFLOW(2) or UNDERFLOW(2).

PL/I raises the ERROR(1) condition if the character string is not a valid arithmetic representation, or if the target data field is insufficient to represent the converted value.

The following examples illustrate various conversions from character to arithmetic data types:

Table 4-1. Character to Arithmetic Conversion

Character	*Target*	*Result*
'00987'	FIXED BINARY(15)	987
'9.87'	FIXED DECIMAL(6,2)	0009.87
'-9.87E2'	FLOAT BINARY(24)	-9.87E2
'-9.87E2'	FIXED DECIMAL(9,2)	0000987.00
'-9.87E2'	FIXED DECIMAL(5,0)	00987
'-987.372'	FIXED DECIMAL(4,2)	ERROR
'2X3'	FIXED BINARY(15)	ERROR

4.3.6 Character to Bit-string Conversion

When performing character to bit-string conversion, the source character string must contain only the characters zero and one. PL/I converts each zero character to a zero-bit, and each one character to a one-bit.

If the target length is greater than the source length, then PL/I pads on the right with zero-bits. If the target length is shorter than the source length, then it truncates on the right. If the source is the null string, or contains all blanks, then the result is a string of zero-bits.

End of Section 4

Section 5
Data Aggregates

An aggregate is a grouping of multiple data items. In PL/I, there are two kinds of aggregates: arrays and structures.

- An array is an ordered collection of data items called elements which all have the same attributes. The elements of an array can be scalar data items or structures. PL/I allows you to reference an entire array by name, or to reference an individual element of an array by using integer subscripts that denote the relative position of the element in the array.

- A structure is a collection of data items called members which can have different data types. The members of a structure can be arrays. PL/I allows you to reference an entire structure by name. You can also reference an individual member of a structure with a qualified reference that gives both the name of the structure and the name of the member.

A variable that represents a data aggregate is called either an array variable or a structure variable.

5.1 Array Declarations

You define an array variable by specifying its attributes in terms of the number of elements in the array and the organization of the elements. These attributes are called the dimensions of the array. The general form of an array variable declaration is.

```
DECLARE name(bound-pair,...) [attribute-list];
```

where name is any valid PL/I identifier. Each bound-pair specifies the number of elements in each dimension of the array and has the following format:

```
[L:]U
```

where L is the lower-bound of the array, and U is the upper-bound. The values L and U can be any integer values such that L is less than or equal to U.

The attribute list is the set of data attributes that apply to all the elements in the array.

The number of elements in each dimension is the extent, and is given by:

(upper bound) - (lower bound) + 1

The total number of elements in an array is the product of the extents of each dimension.

For example, the following statements are equivalent:

```
declare A(3,4) character(2);
declare A(1:3,1:4) character(2);
```

Both statements define an array whose dimension is two, and whose elements are character strings of length two. The extent of the first dimension is three, and the extent of the second dimension is four. Thus you can visualize A as an array with three rows and four columns whose elements are character strings of length two.

	1	2	3	4
1	xx	xx	xx	xx
2	xx	xx	xx	xx
3	xx	xx	xx	xx

Figure 5-1. Two-dimensional Array

The statement

```
declare B(-2:5,-5:5,5:10) fixed binary;
```

defines the array B to be a three-dimensional array whose subscripts range from -2 to 5, -5 to 5, and 5 to 10, respectively. The corresponding extents are eight, eleven, and six respectively. Thus B contains 528 data items of FIXED BINARY data type.

The following rules apply when specifying dimensions in an array:

- In PL/I, there is no formal limit to the number of dimensions an array can have. However, the practical limit is dictated by the total amount of available data storage, and the overall complexity of any expression that you use to reference an individual element within the array.

- All the bounds must be integer constants.
- The lower bound must be less than or equal to the upper bound.
- At run-time, an out-of-bound array reference produces unpredictable results.

5.2 Array References

In PL/I, any reference to an individual array element must be subscripted. The list of subscripts must be enclosed in parentheses. In multidimensional arrays, the number of subscripts must match the number of dimensions.

A subscripted reference to an array element can be any variable or expression that PL/I converts to an integer value. For example:

```
declare scores(20) fixed binary;
declare (counter, total) fixed binary;
total = 0;
do counter = 1 to 20;
   total = total + scores(counter);
end;
```

Figure 5-2 illustrates the concept of subscripted array references.

declare array_A(4) fixed; /* 4 columns */

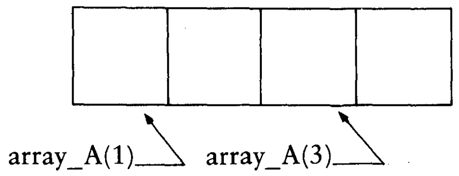
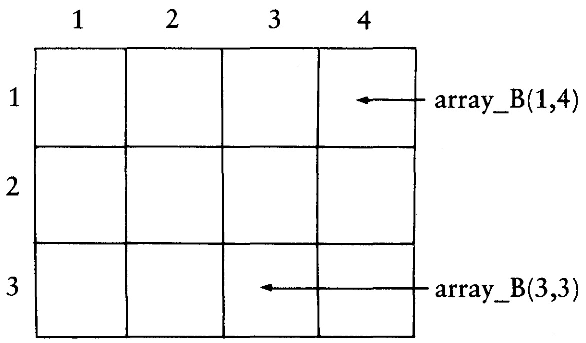


Figure 5-2. Array Element References

Figure 5-2. continues on the following page.

declare array_B(3,4) fixed; /* 3 rows, 4 columns */



declare array_C(2,3,4) fixed; /* 2 planes, 3 rows, 4 columns */

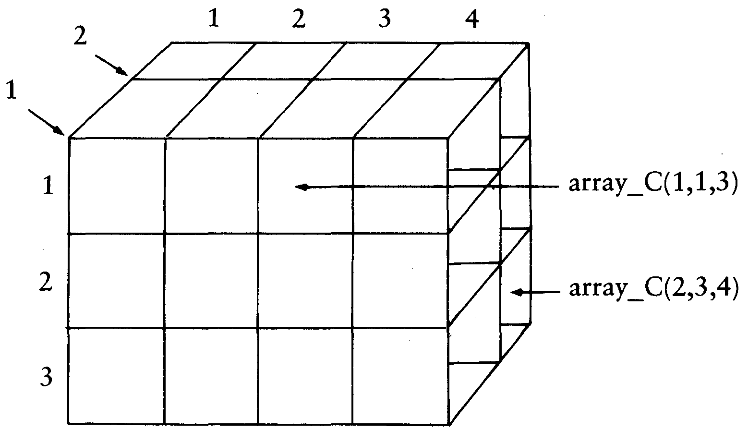


Figure 5-2. Array Element References (continued)

5.3 Initializing Array Elements

You can use the INITIAL attribute with an array declaration to specify values for the elements before execution. For example, the statement

```
declare colors(4) character(10) varying
        static initial ('RED','BLUE','GREEN','YELLOW');
```

assigns a value to each of the elements of the array as shown below:

1	2	3	4
RED	BLUE	GREEN	YELLOW

Figure 5-3. Array Initialization

If you assign each element of an array the same value, the INITIAL attribute can specify an iteration factor in the following format,

INITIAL (value[,value]...);

where the initializing value has the form:

[(iteration-factor)] constant-expression

The iteration factor is an unsigned decimal constant indicating the number of times to use the specified constant. The constant expression can be any reference to an arithmetic or string constant or to the NULL built-in function, and must be compatible with the data being initialized.

For example, the statement

```
declare test_scores(100) fixed binary
        static initial((100)0);
```

initializes all the elements of the array test_scores to zero.

The statement

```
declare grid(15) pointer;
        static initial((15)null);
```

initializes the array grid with null pointer values.

The statement

```
declare numbers(10) character(10)
        static initial((10)'0123456789');
```

initializes all ten elements of numbers with the character string constant '0123456789'.

The statement

```
declare numbers(10) character(10)
        static initial((5)'0123456789',(5)'0');
```

initializes five elements of numbers with the constant'0123456789' and five elements with the constant '0'.

PL/I stores the elements of an array internally in row-major order. That is, the far right subscript varies the most rapidly. If you declare the array with the INITIAL attribute, or reference the entire array in a GET or PUT statement, PL/I accesses the elements in the same order.

For example, using the array declaration

```
declare test_scores(2,2,2) fixed static
                           initial (1,2,3,4,5,6,7,8);
```

PL/I assigns values to the elements in the following order:

```
    test_scores(1,1,1) = 1
    test_scores(1,1,2) = 2
    test_scores(1,2,1) = 3
    test_scores(1,2,2) = 4
    test_scores(2,1,1) = 5
    test_scores(2,1,2) = 6
    test_scores(2,2,1) = 7
    test_scores(2,2,2) = 8
```

PL/I uses the same order to output the elements in a PUT statement, such as:

```
do i = 1 to 2;
   do j = 1 to 2;
      do k = 1 to 2;
            put list(test_scores(i,j,k));
      end;
   end;
end;
```

5.4 Arrays in Assignment Statements

Only in certain restricted cases does PL/I allow an array variable to be the target of an assignment statement. Any statement of the form

array variable A = array variable B;

is valid if the arrays are identical in dimension and data type, and the storage for both arrays is connected. In this case, each element in array-variable-A is assigned the corresponding element in array-variable-B. For example:

```
declare A(20) fixed binary;
declare B(20) fixed binary;
A = B;
```

Individual elements of an array can also be targets of assignment statements. For example, in the following code sequence the elements of one array are assigned values computed using the elements in another array.

```
declare array_A(10) float binary;
declare array_B(10)fixed binary static
        initial (0,1,2,3,4,5,6,7,8,9);
declare i fixed binary;
do i = i to 10;
  array_A(i) = sqrt(array_B(i));
end;
```

Array variables cannot be operands for arithmetic operators such as + and −. For example, any statement of the form,

C = A + B;

is invalid if A, B, and C are array variables.

PL/I does not allow any statement of the forms:

- array-variable = constant;
- array-variable = expression;

For example, the following code sequence is illegal in PL/I:

```
declare A(10) fixed binary;
declare n fixed binary static initial(2);
A = n;
```

However, you can obtain the same effect as follows:

```
declare A(10) fixed binary;
declare n fixed binary static initial(2);
declare i fixed binary;
do i = i to 10;
   A(i) = n;
end;
```

5.5 Structures

A structure is an aggregate that can contain items of different data types. You can use structures to represent data that more closely reflect real-life objects.

The data items contained in the structure are called its members. Structures can contain scalar data items, arrays of scalar items, or other structures called substructures. Structures are ordered hierarchically. The main structure is called the major structure and any substructure is called a minor structure.

A structure declaration defines its organization and the names of the members on each level in the structure. Every structure declaration must contain:

- a name for the major structure,
- the names and data attributes of its members,
- and a level number for each name to define its level in the hierarchical order.

The general form of a structure declaration is:

```
DECLARE [level] name [attribute-list]...
        [,[level] name [attribute-list]];
```

Level numbers precede the names and must be separated from them by one or more spaces. The level number of a major structure is always one. The definitions of each member, including its level number, name, and attributes, must be separated by commas. The level numbers of the members of a minor structure must be greater than the level number of the minor structure.

Note: structure names cannot have data type attributes, but can have a dimension attribute. Structure names can also have the BASED, EXTERNAL, or STATIC attributes.

The following statement is an example of a structure definition:

```
declare 1 bill,
          2 name,
            3 last_name character(20),
            3 first_name character(20),
            3 middle_initial character(1),
          2 address,
            3 street character(20),
            3 city character(10),
            3 state character(3),
            3 zip character(5),
          2 charges,
            3 shop fixed decimal(10,2),
            3 snack_bar fixed decimal(10,2),
            3 misc fixed decimal(10,2),
            3 dues fixed decimal(10,2);
```

Figure 5-4 shows the hierarchy of levels corresponding to this declaration.

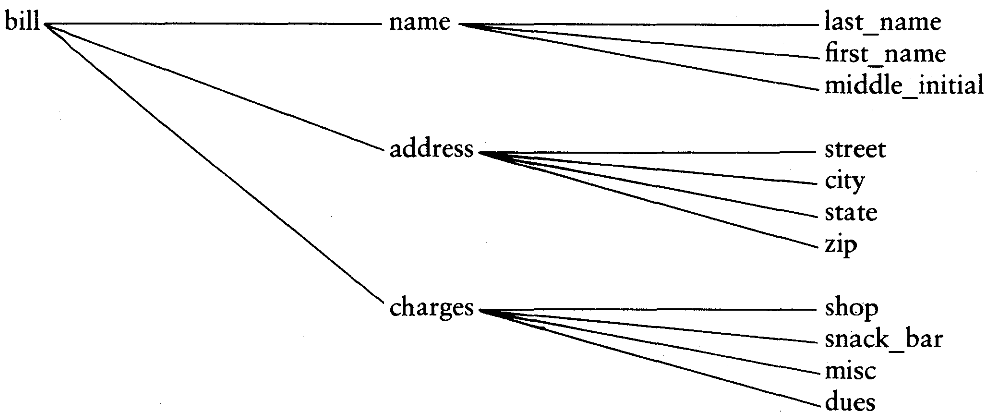


Figure 5-4. Hierarchy of Structure Levels

Ambiguities can arise when referencing the members of structures because the name of a structure member can occur as the name of the member of another structure, or as the name of a data item in a substructure of the same structure. These ambiguities arise only with member names in a common scope of definition.

To resolve such ambiguities, use qualified names to reference members of structures. In a qualified name, the member name is preceded by a list of structure names in ascending order of level number, each followed by a period and zero or more blanks. The only structure names required are those that determine a unique reference to the member name.

For example, in the following structure

```
declare 1 A,
          2 B,
            3 C fixed,
            3 D fixed,
          2 BB,
            3 C fixed,
            3 D fixed;
```

a reference to item C, or D, or A.C, or A.D is ambiguous. The qualified names B.C, or B.D, or BB.C, or BB.D uniquely identify the structure elements. The fully qualified names are:

```
A.B.C
A.B.D
A.BB.C
A.BB.D
```

5.6 Mixed Aggregates

A mixed aggregate is either an array whose elements include structures, or a structure whose members include arrays. You can define an array whose elements are a single type of structure by giving the major structure name a dimension attribute in the structure declaration. You can also give minor structures a dimension attribute. When you declare a structure with a dimension attribute, each member of the structure inherits the dimension and becomes an array.

For example, the statement

```
declare 1 student_list(100),
          2 student_name,
            3 last_name character(10),
            3 first_name character(10),
            3 middle_initial character(1),
          2 social_security_number character(9),
          2 country character(10),
          2 grades(5) character(2);
```

defines an array of structures whose major structure name is student_list. Each structure element of the array has the sub-array grades as a member. To reference an entry in the array, you must use a qualified name together with subscripts for the structure names that have a dimension attribute, and the member name if it has a dimension attribute. The subscripts do not have to appear with their corresponding name, but must occur in parentheses separated by commas and in correct order.

For example, any of the following forms is a fully qualified, unambiguous reference to the third grade entry for the sixty-first entry of the array student_list:

```
student_list(61).grade(3)
student_list.grade(61,3)
student_list(61,3).grade
```

5.7 Mixed Aggregate Referencing

You can reference an entire mixed aggregate by name. A reference to data items inside a mixed aggregate can be partially subscripted, and/or partially qualified. Any such a reference into a mixed aggregate must identify connected storage (see Appendix D). Connected storage means the data elements occupy consecutive storage locations.

For example, consider how PL/I stores the data elements for the following declarations:

```
declare 1 color(100),
          2 hue character(10) varying,
          2 intensity fixed binary;
```

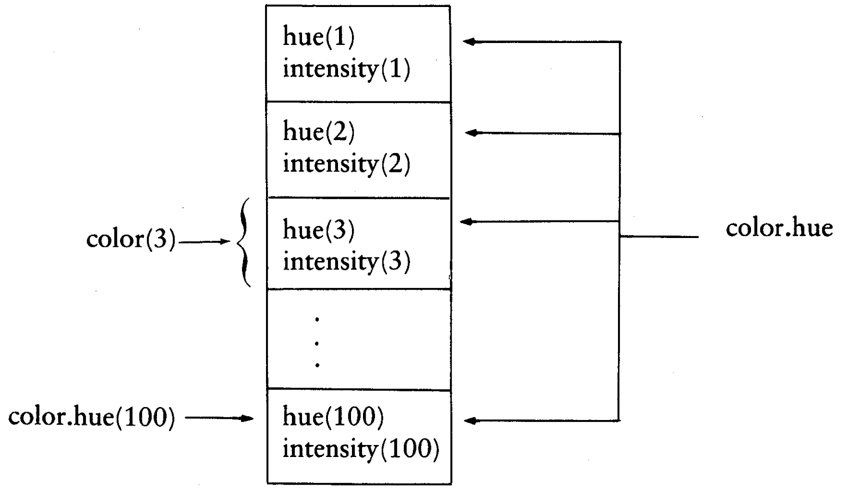


Figure 5-5a. An Array of Structures

Now, the similar declaration,

```
declare 1 color,
          2 hue(100) character(10) varying,
          2 intensity(100) fixed binary;
```

stores the data elements as shown in Figure 5-5b.

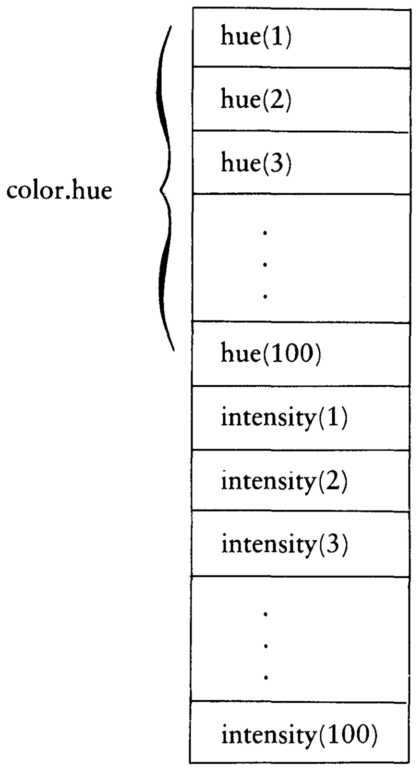


Figure 5-5b. A Structure of Arrays

In Figure 5-5a, color is dimensioned and each of its members hue and intensity inherits the declared dimension. Therefore, each appears as an array, but the elements do not occupy consecutive storage locations.

In Figure 5-5b, color has two members, both of which are dimensioned. The elements of each array occupy consecutive storage locations.

Referring to Figure 5-5a, the storage for color(3), or color.hue(100) is connected storage, but the storage for color.hue is unconnected. However, in Figure 5-5b, the storage for color.hue is connected.

Each type of declaration has its advantages and disadvantages. The specific application and method of access in a program determine the type of declaration, and the storage that results.

End of Section 5

Section 6
Assignments and Expressions

6.1 The Assignment Statement

The assignment statement sets a variable equal to the value of an expression or constant. The assignment statement has the general form,

variable = expression;

where variable is a scalar element, an array, a structure name, or a pseudo-variable (Section 6.8). The assignment statement contains no distinctive keyword.

6.2 Expressions

An expression is any valid combination of operands and operators that PL/I computes at run-time to produce a single value.

Various syntactic rules govern the arrangement of references, operators, and parentheses in an expression. A reference can be a constant, a variable, or a function. An operator defines the computation to perform, using the operands to which it is applied. Parentheses enclose various portions of the expression.

The following sections present the proper formulation of operands, operators, and parentheses.

6.2.1 Prefix Expressions

A prefix expression consists of a unary prefix operator followed by an expression called the operand. PL/I first evaluates the operand and then applies the unary operator to the result.

Two examples of prefix expressions are shown below:

```
^A          (logical Not of A)
-SQRT(B)    (minus square root of B)
```

6.2.2 INFIX Expressions

An infix expression consists of two expressions called operands separated by an infix operator. PL/I first evaluates the operands, that can be expressions, and then applies the operator to the result.

Two examples of infix expressions follow:

```
A+B     (sum of A and B)
C**2    (C squared)
```

6.3 Precedence of Operators

In any unparenthesized expression or subexpression, PL/I applies operators according to a set of precedence rules. Table 6-1 shows the fixed order of precedence from highest to lowest with operators of equal precedence listed on the same line.

Table 6-1. PL/I Operator Precedence

Operator	*Symbol*	*Priority*
Exponentiation	**	1
Logical Not	^ or ˜	1
Prefix Operators	+, −	1
Multiplication, Division	*, /	2
Addition, Subtraction	+, −	3
Concatenation	||, !!, or \\\\	4
Relational Operators	=, ^=, <, ^<, >, ^>, <=, >=	5
Logical And	&	6
Logical Or	|, !, or \\	7

When evaluating an unparenthesized expression, PL/I inserts a balanced parenthesis pair around the highest precedence operators and their corresponding operands first. It continues descending to lower precedence operators and their operands until the entire expression is properly parenthesized.

When equal precedence operators occur at the same level, PL/I evaluates prefix operators and exponentiation from right to left, with the remaining operators evaluated from left to right.

For example, the Compiler treats the unparenthesized expression,

```
2 + Z * X ** Y ** 2 / 5 - Q
```

as the expression:

```
(2 + ((Z * (X ** (Y ** 2))) / 5)) - Q
```

6.4 Concatenation

The infix operator || concatenates either bit strings or character strings. Both operands must be of the same type, and the result is the same type as the operands. The length of the resulting string is always the sum of the lengths of the operands.

For character-string concatenation, if either operand has the VARYING attribute, then the result has the VARYING attribute. For example, the code segment,

```
declare
        A character(3),
        B character(6) varying,
        C character(20);
A = 'ABC';
B = 'ABCDEF';
C = A || B;
```

assigns the character string ABCABCDEF of length nine to the variable C.

6.5 Relational Operators

In PL/I, relational operators are infix operators that compare the relationship between two operands in an algebraic sense. Computational values can be compared according to general algebraic rules, but noncomputational values can only be compared for equality or inequality.

Character string, bit string, and arithmetic data items can be compared using any relational operator, but only the equal and not equal operators can compare ENTRY, LABEL, POINTER, and FILE data items.

ENTRY values are equal only if they identify the same entry point in the same block activation.

LABEL values are equal only if they identify the same statement in the same block activation. A LABEL value that identifies a label on a null statement is not equal to a LABEL value on any other statement.

POINTER values are equal only if they identify the same storage location, or they are both null values.

FILE values are equal only if they identify the same File Parameter Block (see Section 10.4).

If the operands differ in data type, PL/I first converts them to a common type before making the comparison, and then produces a bit string of length one with the value '1'B, true, if the operands are equal, and '0'B, false, if the operands are not equal (see Section 4).

PL/I compares character strings by extending the shorter operand on the right with blanks until it is the same length as the longer operand. It makes the comparison character-by-character from left to right using the ASCII collating sequence (see Appendix C). In this sequence, the value of any upper-case letter is less than any lower-case letter, and the value of any numeric character is less than any alphabetic character.

For example, given the two strings JACK and JACKSON, PL/I first pads the shorter string with blanks and then compares the strings starting on the left as shown below, ƀ denotes a blank:

J 4A	A 41	C 43	K 4B	ƀ 20	ƀ 20	ƀ 20

J 4A	A 41	C 43	K 4B	S 53	O 4F	N 4E

Because S, 53h, is greater than a blank, 20h, JACKSON is greater than JACK.

PL/I compares bit strings by extending the shorter string on the right with zero-bits. Comparison is then made bit by bit from left to right with zero considered less than one. For example, '00010000'B is less than '00010001'B.

6.6 Bit-string Operators

Table 6-2 shows the PL/I bit-string operators.

Table 6-2. PL/I Bit-string Operators

Operator		*Symbol*
Complement	(logical Not)	^ or ~
Inclusive Or	(logical Or)	| or !
And	(logical And)	&

PL/I performs bit-string operations on a bit-by-bit basis. The unary Not operator reverses each bit value in the bit-string operand, changing a zero-bit to a one-bit, and a one-bit to a zero-bit. For example, given the bit string A = '01110010'B, then ^A = '10001101'B.

The Or and And operators require two bit-string operands. If the operands are of unequal length, PL/I extends the shorter one on the right with zero-bits until it is equal in length to the other operand. The resulting string length equals the longer of the two operands.

The Or and And operators follow the rules of Boolean algebra as shown below:

x	y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

Additional Boolean functions are easily constructed using the BOOL built-in function (see Section 13.3).

6.7 Exponentiation

PL/I computes exponentiation as a series of multiplications if the exponent is a nonnegative integer constant. Otherwise, it evaluates the operation using the built-in

LOG and EXP transcendental functions. When evaluating an exponential expression, PL/I treats the following as special cases:

- If X=0 and Y>0, then X**Y = 0.
- If X=0 and Y<0, then the run-time system raises the ERROR(3) condition.
- If X^=0 and Y=0, then X**Y = 1.
- If X<0 and Y is not an integer, then the run-time system raises the ERROR(3) condition.

6.8 Pseudo-variables

SUBSTR and UNSPEC are the names of two PL/I built-in functions (BIFs), that you can use as source operands in expressions. However, you can also use SUBSTR and UNSPEC as the target operands on the left-hand side of assignment statements. In this case SUBSTR and UNSPEC are called pseudo-variables because they appear to act like simple program variables.

6.8.1 Character SUBSTR

The character substring operator allows you to access individual characters in a string. It takes one of the following two forms,

```
char-exp = SUBSTR(char-variable,i)
char-exp = SUBSTR(char-variable,i,j)
```

where char-variable is an optionally subscripted CHARACTER or CHARACTER VARYING variable reference, and i and j are FIXED BINARY expressions.

The first form extracts the substring starting at position i for the remainder of the string, where the first character position is numbered as one.

The second form shown above performs the same function as the first, but the length of the extracted substring is j. The result is undefined if either i or i+j exceeds the string length, where the length is the declared fixed size for CHARACTER variables, and the current length for CHARACTER VARYING variables.

For example, if the variable word contains the character string 'Josephine', then the following assignments result in the strings indicated.

```
x = SUBSTR(word,1);   /* x = 'Josephine'  */
y = SUBSTR(word,5);   /* y = 'phine'      */
z = SUBSTR(word,1,4); /* z = 'Jose'       */
```

When SUBSTR appears as a pseudo-variable on the left of an assignment, it must appear alone. That is, SUBSTR cannot be embedded in a string expression when it serves as the target of a string assignment. SUBSTR appears in this context as:

SUBSTR(char-variable,i) = char-exp;
SUBSTR(char-variable,i,j) = char-exp;

The first form assigns the character expression given by char-exp to the substring in the char-variable, starting at position i, and extending through the length of the char-variable.

The second form has the same effect, except the field width that receives the characters is restricted to length j. The values of i and i+j must be within the current or fixed string length, otherwise the operation produces undefined results.

The same char-variable can appear on both the left and right side of an assignment statement without partial substring overwrite during the assignment.

For example, if the variable word contains the character string 'Collegiate', then following the statement

```
substr(word,7) = substr(word,10,1);
```

the variable contains the string 'Collegeƀƀƀ'.

6.8.2 Bit SUBSTR

In PL/I, bit substring operations are similar to the character SUBSTR shown above, with some restrictions. First, PL/I limits bit strings to the precision range one through sixteen, corresponding to single- and double-byte values. To account for the intermediate precision values during compilation, the length of a bit substring operation must be constant.

Thus, the forms for bit substring are

```
SUBSTR(bit-variable,k)
SUBSTR(bit-variable,i,k)
```

where the bit-variable is an optionally subscripted BIT variable reference; k is an integer constant in the range one to sixteen, and i is a FIXED BINARY expression.

The effect of the SUBSTR operation is identical to the character operation described above, except PL/I selects a bit string of length k when SUBSTR appears in an expression, and assigns it when SUBSTR appears on the left as a target of a bit-string store operation.

The following section gives an example of bit SUBSTR.

6.8.3 UNSPEC

The UNSPEC BIF returns a bit-string value of the internal representation of the argument. The form of the UNSPEC BIF is

```
variable = UNSPEC(argument);
```

where the argument is an optionally subscripted reference to a data item that occupies a single- or double-byte memory location.

Note: PL/I does not allow a temporary result as the argument of UNSPEC.

When UNSPEC appears as a pseudo-variable on the left of an assignment statement, PL/I converts the assigned value to a bit string and directly stores it into the single- or double-byte location of the variable. Thus, UNSPEC allows you to access single- and double-byte variables as if they are 8- and 16-bit string data items.

The UNSPEC pseudo-variable is often used as an escape mechanism when the usual features of the language do not appear to allow access to the underlying facilities. Do not use UNSPEC instead of a more appropriate high-level language facility, because UNSPEC is implementation dependent. In fact, whenever it seems necessary to use UNSPEC, examine the problem in a more general way to see if its use can be avoided.

The following example shows two memory locations being accessed. The UNSPEC operation loads two absolute addresses into two pointer variables. Two based variables, in turn, overlay these two memory locations so they can be accessed as 16- and 8-bit quantities. The bit SUBSTR pseudo-variable is then applied to move a substring from one location to the other.

```
declare
   (P, Q) pointer,
   A bit(16) based(P),
   B bit(8) based(Q),
   I fixed;
      .
      .
      .
I = 4;
unspec(P) = 'FF80'b4;
unspec(Q) = 'FFF0'b4;
   .
   .
   .
substr(B,4,2) = substr(A,I,2);
   .
   .
   .
```

End of Section 6

Section 7 Storage Management

Every variable in a PL/I program is associated with a storage class attribute. The storage class determines how and when PL/I allocates storage for a variable, and whether the variable has its own storage or shares storage with another variable.

PL/I supports three different storage classes:

- STATIC
- AUTOMATIC (the default in PL/I)
- BASED

For the STATIC and AUTOMATIC storage classes, the Compiler allocates storage before execution by generating code that automatically associates the variable name with a given storage location at run-time. For the BASED storage class, the Compiler maintains the variable name and attributes, but does not allocate any storage for it. This allows the run-time system to allocate and free storage during program execution.

To improve internal addressing mechanisms, PL/I treats AUTOMATIC storage in the same way as STATIC storage, except in procedures marked as RECURSIVE.

Storage class attributes are properties of elements, arrays, and major structure variables. Entry names, filenames, or members of data aggregates cannot have these attributes.

7.1 Storage Class Attributes

The STATIC, INITIAL, and AUTOMATIC attributes direct the Compiler to allocate storage and generate code that associates the declared variable name with a given storage location at run-time.

7.1.1 The STATIC Attribute

The Compiler allocates storage for a variable declared with the STATIC attribute before execution of the main procedure. The storage remains allocated until the program ends. Variables belonging to the STATIC storage class can have their data values initialized with the INITIAL attribute.

7.1.2 The INITIAL Attribute

The INITIAL attribute directs the Compiler to assign initial constant values to STATIC data items upon storage allocation. The general form of the INITIAL attribute is

INITIAL (value[,value] ...)

where value has the form:

[(iteration-factor)] constant-expression

The optional iteration factor is an integer that specifies the number of times the constant is repeated. The constant expression must be a literal constant value that is compatible with the data type being initialized. It consists of either an optionally signed arithmetic constant, a string constant, or a NULL pointer value.

You can initialize array data items with a single statement. The statement must begin with the first element of the array, and continue in row-major order until the end of the set of initialized constants. The number of constants should not exceed the size of the initialized array. Structure members must be individually initialized.

Assignments of constants to variables follow the rules for assignment statements. For example, PL/I does blank padding on the right when a shorter character string is assigned to a longer character string variable.

Note: only STATIC variables can have the INITIAL attribute to be compatible with the ANSI Subset G PL/I standard.

The following code sequence illustrates the STATIC and INITIAL attributes:

```
declare A fixed binary static initial(0);
declare B(8) character(2) initial((8)'AB') static;
declare
        1 fcb static,
          2 fcb_drive fixed(7) initial(0),
          2 fcb_name  character(8) initial('EMP'),
          2 fcb_type  character(3) initial('DAT'),
          2 fcb_ext   bit(8)     initial('00'B4),
          2 fcb_fill(19) bit(8);
```

7.1.3 The AUTOMATIC Attribute

Usually, the AUTOMATIC attribute forces data storage allocation upon entry to the PROCEDURE or BEGIN block in which the variable appears. In PL/I, AUTOMATIC storage is statically allocated to improve variable addressing and execution speed.

The only exception is in the case of recursion, where the AUTOMATIC variables must use the dynamic storage mechanism to prevent data overwrite on recursive calls.

7.2 Based Variables and Pointers

A based variable is a variable that describes storage that must be accessed with a pointer (see Section 3.4). The pointer is the location where the storage for the based variable begins, and the based variable itself determines how PL/I interprets the contents of the storage beginning at that location. Thus the pointer and the based variable together are essentially equivalent to a nonbased variable.

You can visualize a based variable as a template that overlays the storage specified by its base. Thus a based variable and pointer can refer to storage allocated for the based variable itself, or to storage allocated for other variables.

The format of the BASED variable declaration is

```
DECLARE name BASED [(pointer-reference)];
```

where the pointer reference is an unsubscripted POINTER variable, or a function call, with zero arguments, that returns a POINTER value.

A pointer-qualified reference can be either implicit or explicit. When you declare a variable as BASED without a pointer reference, each reference to the variable in the program must include an explicit pointer qualifier of the form,

```
pointer-exp -> variable
```

where pointer-exp is a pointer-valued expression.

When you declare a variable as BASED with a pointer reference, you can reference it without a pointer-qualifier. The run-time system reevaluates the pointer reference at each occurrence of the unqualified variable using the pointer expression given in the variable declaration.

The following example illustrates the difference between explicit and implicit pointer-qualified reference.

```
Main:
    procedure options(main);
    declare
       list_A(100) fixed binary based,
       list_B(100) fixed binary based(list_B_ptr),
       (list_A_ptr,list_B_ptr) pointer;
               .
               .
               .
    list_A_ptr -> list_A(47) = 0;   /*  explicit reference */
    list_B(47) = 0;   /* implicit reference */
               .
               .
               .
end main;
```

You can declare the same pointer name in a different environment, and use it to make an *implicit* pointer-qualified reference. However, PL/I takes the pointer variable name or pointer-valued function name given in the pointer reference from the scope of the original BASED declaration. The following example illustrates this concept.

```
A:
   procedure options(main);
   declare
      (i,j) fixed binary,
      p pointer,
      x fixed binary based(p);
   p = addr (i);
   x = 2; /* implicit reference; x refers to i */
B:
   procedure;
   declare
      p pointer; /* local to B */
   p = addr(j);
   x = 12; /* implicit reference; x still refers to i */
   p -> x = 3; /* explicit reference; x now refers to j */
end A;
end B;
```

The following statements are examples of valid BASED variable declarations:

```
declare A character(8) based;
declare B pointer based(Q);
declare C fixed based(P);
declare D bit(8) based(F());
```

7.3 The ALLOCATE Statement

The ALLOCATE statement explicitly allocates storage for a variable with the BASED attribute. The general form of the ALLOCATE statement is:

ALLOCATE based-variable SET(pointer-variable);

The ALLOCATE statement directs PL/I to obtain a segment of storage from the dynamic storage area that is large enough to hold the value of the based variable. If a segment of the requested size is not available, the run-time system signals ERROR(7).

The based variable must be an unsubscripted variable reference, where the variable is declared with the BASED attribute in the scope of the ALLOCATE statement. The run-time system stores the allocation address into the pointer variable named in the SET clause.

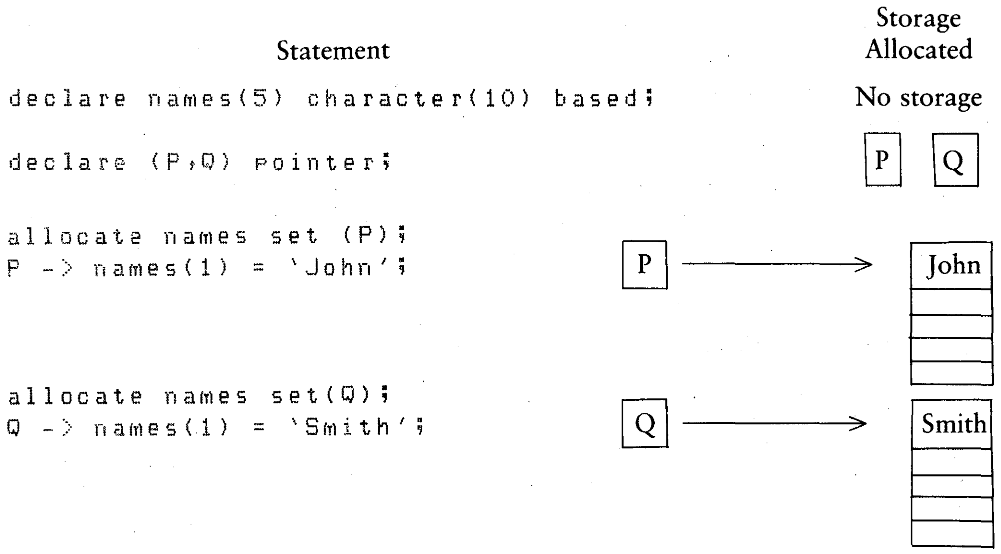
Storage allocated in this manner remains allocated until a corresponding FREE statement is executed, using the allocation address held by the pointer variable as an operand.

7.4 Multiple Allocations

The ALLOCATE statement allocates storage each time it is executed in the program. A program can allocate storage for a single based variable more than once, and as long as each allocation has a unique pointer, the program can reference all of them. For example:

```
declare names(5) character(10) based;
declare (P,Q) pointer;
allocate names set(P);
P -> names(1) = 'John';
        .
        .
        .
allocate names set(Q);
Q -> names(1) = 'Smith';
        .
        .
        .
```

In this example, there is no compile-time storage allocation for the array variable names. The Compiler automatically allocates storage for the pointers P and Q at compile time. At run-time, the ALLOCATE statements obtain two different allocations for names that can then be referenced with the appropriate pointer. Figure 7-1 illustrates this concept.



```
declare names(5) character(10) based;

declare (P,Q) pointer;

allocate names set (P);
P -> names(1) = 'John';

allocate names set(Q);
Q -> names(1) = 'Smith';
```

Figure 7-1. Multiple Allocations of a BASED Variable

Note: when multiple allocations of a based variable all have the same pointer, the pointer only references the most recent allocation, and not any preceding ones.

7.5 The FREE Statement

A BASED variable remains allocated until released with the FREE statement. The general form of the FREE statement is

FREE [pointer-variable ->] based-variable;

where the pointer variable addresses an allocation of storage that must have been previously obtained from the dynamic storage area using the ALLOCATE statement. Unpredictable results can occur if a program attempts to free unallocated storage.

If the pointer variable is not given in the FREE statement, then the based variable must be declared with the pointer reference option. In this case, the run-time system returns the storage addressed by the pointer reference to the dynamic storage area.

The run-time subroutines that maintain the dynamic storage area automatically coalesce contiguous storage segments as they are released using the FREE statement.

Note: when the FREE statement releases a storage allocation, both the pointer and the contents of the storage area become undefined. Unpredictable results can occur if the program makes any subsequent reference to the freed storage.

The following code sequence illustrates the FREE statement:

```
declare
        (P, Q, R) pointer,
        A character(10) based,
        B fixed based(R);
                .
                .
                .
allocate A set(P);
allocate B set(R);
allocate A set(Q);
           .
           .
           .
free P -> A;
free Q -> A;
free B;
```

7.6 The NULL BIF

The NULL BIF returns a pointer value that is a unique nonvalid storage address. This unique address is useful in marking various pointer values as empty, and is especially useful in the construction of a linked list.

A linked list is a data structure composed of elements that not only contain a data area but also contain a pointer to the next element in the list. In such a list, the last element has no following element, and its pointer has an invalid, null, value. Figure 7-2 shows a linked list.

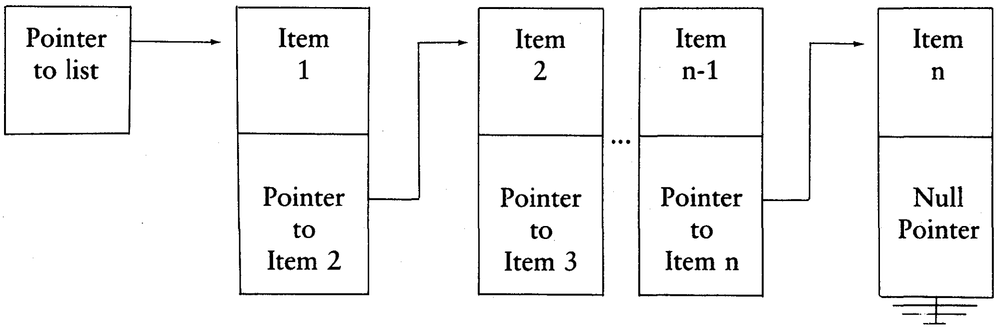


Figure 7-2. Linked List

The general form of the NULL function call is:

NULL[()]

Pointer values do not necessarily begin with a null value when program execution begins. However, pointer values can be given a null value by using the value returned by NULL in the variable declaration INITIAL option.

NULL is an invalid pointer qualifier for a based variable. For example, the following code sequence is invalid in PL/I:

```
declare A pointer;
declare list(10) fixed binary based(A);
        .
        .
        .
A = null();
A ->list(10) = 32767;    /*  this is invalid!!*/
```

Section 10 in the *PL/I Language Programming Guide* contains sample programs that illustrate the use of BASED variables and the NULL function.

7.7 The ADDR BIF

The ADDR BIF returns a pointer value that addresses the segment of memory occupied by the variable name given as the argument. The form of the ADDR BIF is:

ADDR(variable name)

Note: the variable name must have an assigned memory address, and cannot be a temporary result created through the application of functions and operators, nor can it be a constant or a named constant such as a FILE, ENTRY, or LABEL constant.

7.8 Storage Sharing

Use of BASED variables in conjunction with the ADDR BIF allows storage sharing in PL/I. With storage sharing, the based variable is not explicitly given storage with the ALLOCATE statement. Rather, the based variable acts as a template that overlays the storage for an existing variable.

To share storage, you must use the ADDR BIF to set the pointer base for the based variable to the address of the existing variable. Subsequent access to the based variable then accesses the overlayed variable. The only requirement is that the length of the based variable, in bits, be less than or equal to the length of the existing variable, in bits.

The program below illustrates storage sharing. Here, the value of a character string is overlayed by a bit-string vector. The output from the program is the character-string value, written in hexadecimal bit-string form.

```
declare
  i fixed binary,
  ptr pointer,
  word character(8),
  bit_vector(8) bit(8) based(ptr);
ptr = addr(word);
get list(word);
do i = 1 to 8;
   put edit(bit_vector(i)) (x(2),b4(2));
end;
```

If you enter the word Digital at the console, the storage location allocated for the variable word appears as shown below:

D	i	g	i	t	a	l	sp

The based variable bit_vector is simply a template that overlays the storage for word as shown below,

xxxxxxxx D	xxxxxxxx i	xxxxxxxx g	xxxxxxxx i	

where x denotes a single bit. Thus, on output, the program reads the bit string starting at the location of word and converts it to a hexadecimal representation of the individual characters stored in word.

Note: there is an important consideration involved in this type of storage sharing. The preceding example depends on knowledge of the internal data representation used by PL/I, namely that eight bits represent a character. Thus, the program is implementation dependent. This runs counter to the Subset G philosophy of writing transportable programs. PL/I allows such storage sharing using based variables, but the resulting code might not be transportable to a different implementation.

7.9 Programming Considerations

Based variables and pointers are powerful tools because they give you direct access to memory. However, use them with caution. Remember that storage obtained with the ALLOCATE statement remains allocated until it is freed or the program ends. Any based variables and pointers that refer to the allocated storage remain active only as long as the block in which they are declared remains active. When control passes out of the block, the storage becomes inaccessible.

Note: PL/I cannot tell if the size of a based variable does not correspond to the size of the storage it refers to. If a program assigns a value to a pointer-qualified reference whose size does not match the allocated storage, then the contents of adjacent storage locations can be destroyed.

The following errors are common when using based variables and pointers:

- assigning a pointer the NULL value somewhere in the program and subsequently using it elsewhere in a pointer-qualified reference.
- using a pointer to reference a based variable whose storage has been freed.
- using a pointer whose value has been lost because of the deactivation of the block in which it was declared.

End of Section 7

Section 8
Sequence Control

PL/I program statements usually execute sequentially. You can use sequence control statements to alter this normal flow with conditional and unconditional branching and controlled looping, as discussed below. Procedure invocations including function calls also alter the normal execution sequence, and are thus considered sequence control statements (see Section 2.5).

8.1 The Simple DO Statement

A DO-group is a sequence of statements that begins with a DO statement and ends with an END statement. The statements must be executable. A DO-group cannot contain any declarations. A DO-group can occur in one of two forms: the simple, noniterative, DO, and the controlled, iterative, DO.

The simple DO statement has the form:

```
[label:]
  DO;
     statement-1;
          .
          .
          .
     statement-n;
  END [label];
```

where Statement-1 through Statement-n constitute the body of the group.

The following code sequence illustrates the simple DO-group:

```
  .
  .
  .
do;
  x = 3.14/2;
  y = sin(x);
  z = 5 + y;
end;
  .
  .
  .
```

8.2 The Controlled DO Statement

The controlled DO statement has one of two general forms:

DO WHILE (condition);
DO control-variable = do-specification;

where the control-variable is a scalar variable; the condition is a Boolean expression, and the do-specification is one of the following:

[start-exp [TO end-exp] [BY incr-exp]] [WHILE(condition)]
[start-exp [BY incr-exp] [TO end-exp]] [WHILE(condition)]
[start-exp [REPEAT repeat-exp]] [WHILE(condition)]

In these general forms, start-exp is an expression specifying the initial value of the control variable; end-exp is an expression representing the terminal value of the control variable; incr-exp is an expression added to the control variable after each execution of the loop, and the repeat-exp is the expression that is assigned to the control variable after each iteration. Condition is an expression yielding a bit-string value that is considered true if any of the bits in the string are one-bits.

If the TO end-exp form is included but the BY incr-exp is omitted, then PL/I assumes the incr-exp to be one. The two forms using TO and BY execute in exactly the same manner, and differ only in the order of these two elements in the program text.

PL/I evaluates the WHILE expression before executing the DO-group. If the condition is false, the loop execution terminates, and control passes to the statement following the balanced END statement.

With the exception of the REPEAT option, PL/I evaluates expressions in the do-specification before executing the loop, so that changes made to the start, end, or incremental values do not affect the number of times a loop executes.

In the case of the REPEAT option however, PL/I recomputes the repeat-exp after each iteration. It then stores this recomputed expression in the control variable and evaluates the WHILE test, if included.

To properly define the actions of iterative groups, PL/I separates them into a sequence of equivalent IF and GOTO statements. In the separation, expressions e1, e2, e3, and e4 are appropriate start-exp, end-exp, incr-exp, repeat-exp, and condition values, while i represents a valid control variable.

The DO WHILE statement

```
DO WHILE (e1);
      .
      .
      .
END;
```

can be expressed as the equivalent sequence of statements:

```
LOOP:
     IF ^e1 THEN
        GOTO ENDLOOP;
           .
           .
           .
        GOTO LOOP;
ENDLOOP:;
```

Similarly, the DO REPEAT group

```
DO  i = e1 REPEAT (e2);
      .
      .
      .
END;
```

becomes:

```
        i = e1;
LOOP:
        .
        .
        .
        i = e2;
        GOTO LOOP;
```

Note: in this case, the loop proceeds indefinitely until terminated by an embedded GOTO or STOP statement.

The WHILE option can be added,

```
DO i = e1 REPEAT (e2) WHILE (e3);
      .
      .
      .
END;
```

resulting in the equivalent statements:

```
        i = e1;
LOOP:
        IF ^e3 THEN
            GOTO ENDLOOP;
            .
            .
            .
       i = e2;
       GOTO LOOP;
ENDLOOP:;
```

The simple iterative DO-group

```
DO i = e1 TO e2;
       .
       .
       .
END;
```

is treated as,

```
DO i = e1 TO e2 BY e3;
      .
      .
      .
END;
```

where e3 = 1, that can be expressed as the equivalent sequence,

```
            i = e1;
         LAST = e2;
         INCR = e3;
LOOP:
         IF endtest THEN
            GO TO ENDLOOP;
            .
            .
            .
         i = i + INCR;
         GOTO LOOP;
ENDLOOP:;
```

where the IF statement containing the endtest compares the control variable with the value of LAST. The comparison is based on the sign of the incrementing value INCR. If INCR is negative, the END-test is:

```
IF i < LAST THEN
   GOTO ENDLOOP;
```

Otherwise, the END-test becomes:

```
IF i > LAST THEN
   GOTO ENDLOOP;
```

Finally, the addition of the WHILE option in,

```
DO i = e1 TO e2 BY e3 WHILE (e4);
      .
      .
      .
END;
```

produces the equivalent sequence:

```
          i = e1;
       LAST = e2;
       INCR = e3;
LOOP:
       IF ^e4 THEN
          GOTO ENDLOOP;
       IF endtest THEN
          GOTO ENDLOOP;
          .
          .
          .
       i = i + INCR;
       GOTO LOOP;
ENDLOOP:;
```

In these equivalent sequences, the value of LAST and INCR takes on the characteristics of the expressions e2 and e3. Arithmetic conversions and comparisons take place at each step according to PL/I rules.

8.3 The IF Statement

The IF statement allows conditional execution of a statement or statement group, based upon the true or false value of a Boolean test. The optional ELSE clause provides an alternative statement or group of statements to execute when the Boolean test produces a false value.

The general form of an IF statement is

```
IF test-condition THEN action-1 [ELSE action-2]
```

where the test-condition is a scalar expression that yields a bit-string value. Action-1 and action-2 can be either simple statements, or compound statements contained within a DO-group or BEGIN block. If either action-1 or action-2 is a simple statement, it cannot be a DECLARE, END, ENTRY, FORMAT, or PROCEDURE statement. The statements in action-1, and action-2 if included, must terminate with a semicolon; therefore the semicolon is not included in the general statement form shown above.

PL/I evaluates the test-condition to produce a bit string. If any bit in the string is equal to one, then PL/I performs action-1. Otherwise, control passes to action-2, if included, or to the next statement in sequence following the IF statement.

IF statements can be nested, in which case PL/I pairs each ELSE with the innermost unmatched IF THEN pair. You can use null statements to force the desired IF ELSE pairing. For example, in the following code segment containing nested IF statements, the null statement following the second ELSE corresponds to the second IF THEN test.

```
┌if A = Y then
│  ┌if Z = X then
│  │  ┌if W > B then
│  │  │   C = 0;
│  │  └else C = 1;
│  └else;
└else A = Y2;
```

8.4 The STOP Statement

The STOP statement unconditionally ends the program, closes all open files, and returns control to the operating system. You can use the STOP statement anywhere in a program to effect a premature termination.

The form of the statement is simply:

STOP;

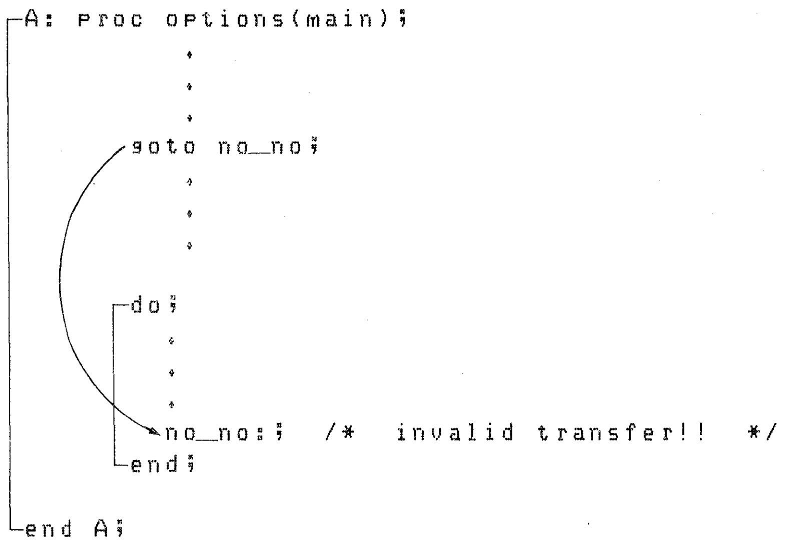
8.5 The GOTO Statement

The GOTO statement unconditionally transfers control to a specific labeled statement. It has the general form,

GOTO label constant | label variable;

where the label constant is a literal label that appears as the prefix of some labeled statement, and label variable is a simple or subscripted label variable that is assigned the value of a label constant.

The evaluated label constant must label a statement in the scope of the GOTO statement, and cannot be within an embedded DO-group of any sort. The following example illustrates this kind of invalid transfer.



```
A: proc options(main);
          .
          .
          .
     goto no_no;
          .
          .
          .
     do;
        .
        .
        .
       no_no;   /*  invalid transfer!!   */
     end;
end A;
```

Transferring control with a GOTO statement is valid only when the target label is known in the block containing the GOTO statement. Thus, transfer of control using GOTO statements and labels is limited to the currently active block or a containing block.

Three examples of valid GOTO statements follow:

```
goto lab1;
goto where;
go to L(J);
```

8.6 The Nonlocal GOTO Statement

In a nonlocal GOTO statement, the evaluated target label constant occurs outside the innermost block containing the GOTO statement.

Note: PL/I programs should usually avoid the nonlocal GOTO because it makes the program harder to debug and maintain.

There are situations when the nonlocal GOTO is appropriate. With terminal error conditions, it is often useful to branch directly to a global error recovery label where program execution recommences. In such a case, PL/I automatically reverts all embedded ON-units and discards procedure return information.

The following code sequence shows an instance of a nonlocal GOTO from within a procedure definition:

```
P:
   procedure;
   goto L;
end P;
call P;
L:;
```

End of Section 8

Section 9
Condition Processing

PL/I supports run-time interception of conditions that would usually end the program. The ON, REVERT, and SIGNAL statements provide this facility.

A condition is any occurrence that interrupts the program's usual flow of execution. A condition can be signaled by the run-time system or by the program itself, at which point control passes to a preestablished logical unit for that condition.

Certain conditions are fatal. This means that the specified logical unit cannot return control to the point where the condition was signaled, but instead must execute a GOTO to a nonlocal label. Other conditions are nonfatal, so that the unit can perform some local action and then return control to the point of the signal.

PL/I recognizes the following general categories of conditions:

- a general error condition (ERROR)
- arithmetic error conditions such as
  - FIXEDOVERFLOW
  - OVERFLOW
  - UNDERFLOW
  - ZERODIVIDE
- I/O conditions such as
  - ENDFILE
  - ENDPAGE
  - KEY
  - UNDEFINEDFILE

9.1 The ON Statement

The ON statement defines the action to take when the specified condition is signaled.

The ON statement has the general form,

    ON condition ON-unit;

where the condition can be one of the conditions listed above.

An ON-unit is *enabled* when it is ready to intercept a condition. An ON-unit is *active* when it is processing a signaled condition. An ON-unit can be a PL/I statement, or several PL/I statements contained in a BEGIN block. PL/I processes the ON-unit when the particular condition named in the ON statement is signaled.

Exit from an ON-unit cannot be through a RETURN statement, although this restriction does not preclude a procedure definition within a BEGIN block.

Once all the statements of the ON-unit are executed, the flow of control resumes at the point where the condition was signaled, provided that the condition is nonfatal. Alternatively, the ON-unit can execute a nonlocal GOTO and transfer control to some label outside the ON-unit.

An ON-unit remains active until its encompassing block ends, or its corresponding REVERT statement executes (see Section 9.3). You can establish more than one ON-unit in the same block. If you establish the same ON-unit in an embedded block, PL/I pushes it onto a condition stack.

```
A:
  procedure options(main);
      .
      .
      .
    on endfile
    on endpage
    on error(1)
      .
      .
      .
    B:
        procedure;
        .
        .
        .
      on error(1)
        .
        .
        .
      revert error(1)
        .
        .
        .
    end B;
      .
      .
      .
A:end A;
```

Figure 9-1. ON-unit Activation

In the preceding example, PL/I first enables the ON-units for the ENDFILE, ENDPAGE, and ERROR(1) conditions. At this point, there are three enabled ON-units. As control flows into the procedure B, PL/I enables the second ON-unit for the ERROR(1) condition and stacks it. There are now four enabled ON-units. Executing the REVERT statement deactivates the most current ON-unit for ERROR(1), the one inside B, and reestablishes the one in the outer, main, procedure block. This again leaves three enabled ON-units.

Note: PL/I allows a maximum of sixteen ON conditions, stacked or otherwise, to be enabled at any given point in the execution sequence. Overflowing the Condition stack is a fatal error. The run-time system stops processing, and prints the message:

```
Condition Stack Overflow
```

9.2 The SIGNAL Statement

The SIGNAL statement causes a particular condition to occur programmatically and invokes a corresponding ON-unit, if one is enabled. If no ON-unit for the condition is enabled, the PL/I default action occurs. If the condition is fatal the default action prints a traceback and terminates the program.

The general form of the SIGNAL statement is

SIGNAL condition;

where the condition is one of the conditions listed above. For example, the statement

```
signal zerodivide;
```

invokes the current ZERODIVIDE ON-unit.

9.3 The REVERT Statement

The REVERT statement disables the current ON-unit for a specific condition and reestablishes the one that preceded it, if it exists.

The general form of the REVERT statement is

REVERT condition;

where the condition is one of the conditions listed above.

For example, the statement

```
revert overflow;
```

disables the current ON-unit for the OVERFLOW conditon.

Note: upon exit from a PROCEDURE or BEGIN block, PL/I automatically reverts any ON-units enabled within the block.

9.4 The ERROR Condition

The ERROR condition is the broadest category of all PL/I conditions. It includes through its subcodes, both system-defined and program-defined conditions. There are four groups of ERROR condition subcodes:

(A)	0	—	63	Reserved for PL/I	(Fatal)
(B)	64	—	127	Program-Defined	(Fatal)
(C)	128	—	191	Reserved for PL/I	(Nonfatal)
(D)	192	—	255	Program-Defined	(Nonfatal)

The general forms of the ON statement with the ERROR condition are

```
ON ERROR[(integer-expression)] on-body;
SIGNAL ERROR[(integer-expression)];
REVERT ERROR[(integer-expression)];
```

where integer expression is a specific subcode in the range 0-255. For example, the statement

```
ON ERROR3) ... ;
```

intercepts the ERROR condition with the subcode 3.

The forms

```
ON ERROR on-body;
ON ERROR(0) on-body;
```

intercept any error condition, regardless of the subcode. Table 9-1 shows the codes currently assigned in group A.

Table 9-1. ERROR Codes - Group A

Error	*Meaning*
ERROR(1)	Data Conversion: Data types do not conform during assignment, computation, or input processing.
ERROR(2)	I/O Stack Overflow. The run-time I/O stack has exceeded sixteen simultaneous nested I/O operations.
ERROR(3)	Invalid argument to function f where f is ACOS, ASIN, LOG, LOG2, LOG10, SQRT, or TAN.
ERROR(4)	I/O Conflict: The attributes of an open file do not match the attributes required for a particular GET, PUT, READ, or WRITE statement.
ERROR(5)	Format stack overflow: Nested format evaluation exceeds 32 levels.
ERROR(6)	Invalid format item: Data item does not conform to format item, or unrecognized item encountered.
ERROR(7)	Free space exhausted: No more space is available in dynamic storage area (TPA).
ERROR(8)	Overlay: No file d:filename.OVR. The indicated file could not be found.
ERROR(9)	Overlay: drive d:filename.OVR. An invalid drive code was passed as a parameter to ?ovlay.
ERROR(10)	Overlay: size d:filename.OVR. The indicated overlay would overwrite the PL/I stack and/or free space if it were loaded.
ERROR(11)	Overlay: nesting d:filename.OVR. Loading the indicated overlay would exceed the maximum nesting depth.
ERROR(12)	Overlay: Read d:filename.OVR. Disk read error during the overlay load, probably caused by premature EOF.

Table 9-1. (continued)

Error	*Meaning*
ERROR(13)	Invalid OS Version. Caused by any operation that generates an operating system call not supported under the current operating system.
ERROR(14)	Unsuccessful Write. Caused by any unsuccessful write operation on a file due to lack of directory space, or lack of disk space.
ERROR(15)	File Not Open. Caused by any attempt to lock or unlock a record in a file that is not open.
ERROR(16)	File Not Keyed. Caused by any attempt to lock or unlock a record in a file that does not have the KEYED attribute.

The following code sequence shows a simple example of the ERROR condition:

```
on error(1)
   begin;
   put skip list('Invalid Input:');
   goto retry;
   end;
retry:
     get list(x);
```

The GET statement reads variable x from the SYSIN file, and if the data is invalid, the run-time system signals ERROR(1). In this case, control passes to the ON-body, which writes an error message to the console, and recommences execution at the retry label.

You can use the SIGNAL statement with the ON statement to intercept either fatal or nonfatal conditions. For example, the statement

```
signal error(64);
```

signals the ERROR(64) condition, and if there is an ON-unit enabled for ERROR(64), then the corresponding ON-body receives control. Otherwise, the program ends with an error message. The statement

```
signal error(255);
```

performs a similar action except that the program does not end if an ON-unit for the ERROR(255) condition is not enabled.

9.5 Arithmetic Error Conditions

PL/I handles several arithmetic error conditions. These conditions are

- FIXEDOVERFLOW[(i)]
- OVERFLOW[(i)]
- UNDERFLOW[(i)]
- ZERODIVIDE[(i)]

where i is an optional integer expression denoting a specific error subcode. Similar to the ERROR function, the ON, REVERT, and SIGNAL statements can specify any of the preceding conditions.

If you do not specify an integer expression, PL/I assumes a zero value. An ON statement with subcode of zero intercepts any subcode from 0-255. PL/I divides the arithmetic condition subcodes into system-defined and program-defined values, analogous to the ERROR function.

Note: PL/I considers all arithmetic error conditions to be fatal. When setting an ON condition for an arithmetic exception, the ON-body should transfer control to a global label. Otherwise, the program ends upon return from the ON-unit.

Table 9-2 shows the system codes for arithmetic error conditions.

Table 9-2. Arithmetic Error Condition Codes

Condition	*Meaning*
FIXEDOVERFLOW(1)	Decimal Add, Multiply, or Store
OVERFLOW(1)	Floating-point operation

Table 9-2. (continued)

Condition	*Meaning*
OVERFLOW(2)	Float Precision Conversion
UNDERFLOW(1)	Floating-point operation
UNDERFLOW(2)	Float Precision Conversion
ZERODIVIDE(1)	Decimal Divide
ZERODIVIDE(2)	Floating-point Divide
ZERODIVIDE(3)	Integer Divide

9.6 The ONCODE BIF

The ONCODE BIF returns a FIXED BINARY value representing the type of error that signaled the most recent condition. If a signal is not active, ONCODE returns a zero. After an ON-unit is activated, ONCODE can determine the exact source of the error. The following code sequence illustrates the use of ONCODE.

```
on error
   begin;
   declare
      code fixed;
   code = oncode();
   if code = 1 then
      do;
        put list('Bad Input:');
        goto retry;
      end;
   put list('Error#',code);
   end;
retry:
```

9.7 Default ON-units

PL/I has default ON-units for each of the condition categories that usually output an appropriate error message and end the program. PL/I does not signal the FIXEDOVERFLOW condition for FIXED BINARY overflow, although it does if FIXED DECIMAL computations exceed their allocated field sizes.

9.8 I/O Conditions

During I/O processing, the run-time system can signal several conditions relating to the access of a particular file. These conditions are

- ENDFILE(file-reference)
- UNDEFINEDFILE(file-reference)
- KEY(file-reference)
- ENDPAGE(file-reference)

where file-reference denotes a file-valued expression. The file value that results need not denote an open file. Section 10.5 describes each of the I/O conditions in detail.

End of Section 9

Section 10
Input/Output Processing

PL/I provides a device-independent input/output system that allows a program to transmit data between memory and an external device such as a console, a line printer, or a disk file.

The collection of data elements transmitted to or from an external device is called the data set. A corresponding internal file constant or variable is called a file.

As with other data items, you must declare all files before you use them in a program. The general form of a file declaration is

```
DECLARE file_id FILE [VARIABLE];
```

where file_id is the file identifier. The declaration defines a file constant if you do not include the VARIABLE attribute. Include the VARIABLE attribute, and the declaration defines a file variable that can take on the value of a file constant through an assignment statement. I/O operations on file variables are valid only after you assign a file constant to a file variable.

Note: by default, PL/I assigns the EXTERNAL attribute to a file constant. Unless you declare a file variable as EXTERNAL, PL/I treats it as local to the block where you declare it.

10.1 The OPEN Statement

PL/I requires that a file be open before performing any I/O operations on the data set. You can open a file explicitly, by using the OPEN statement, or implicitly by accessing the file with one of the following I/O statements:

- GET EDIT
- PUT EDIT
- GET LIST
- PUT LIST
- READ
- WRITE
- READ Varying
- WRITE Varying

Sections 11 and 12 contain detailed descriptions of the various I/O statements.

The general form of the OPEN statement is

```
OPEN FILE(file_id) [file-attributes];
```

where file_id is the identifier that appears in a FILE declaration statement, and file-attributes denotes one or more of the following PL/I keywords:

- STREAM | RECORD
- PRINT
- INPUT | OUTPUT | UPDATE
- SEQUENTIAL | DIRECT
- KEYED
- TITLE
- ENVIRONMENT
- PAGESIZE
- LINESIZE

Multiple attributes on the same line are conflicting attributes so you can only specify one. The first one listed is the default attribute. The default values for the other attributes are:

```
TITLE('file_id.DAT')
ENVIRONMENT(Buff(128))
LINESIZE(80)
PAGESIZE(60)
```

Note: All the attributes are optional and you can specify them in any order.

A STREAM file contains variable length ASCII data. You can visualize it as a sequence of ASCII character data, organized into lines and pages. Each line in a STREAM file is determined by a linemark, that is a line-feed or a carriage return line-feed pair. Each page is determined by a pagemark, or form-feed. ED automatically inserts a line-feed following each carriage return, but files that PL/I creates can have line-feeds without preceding carriage returns. PL/I then senses the end of the line when it encounters the line-feed.

A RECORD file contains binary data. PL/I accesses the data in blocks determined by a declared record size, or by the size of the data item you use to access the file. A RECORD file must also have the KEYED attribute, if you use FIXED BINARY keys to directly access the fixed-length records.

PRINT applies only to STREAM files, and indicates that the data is for output on a line printer.

For an INPUT file, PL/I assumes that the file already exists when it executes the OPEN statement. When it executes the OPEN statement for an OUTPUT file, PL/I creates the file. If the file already exists, PL/I first deletes the old one, then creates a new file.

You can read from and write to an UPDATE file. PL/I creates an UPDATE file if it does not exist when executing the OPEN statement. An UPDATE file cannot have the STREAM attribute.

SEQUENTIAL files are accessed sequentially from beginning to end. DIRECT files are accessed randomly using keys. PL/I automatically gives DIRECT files the RECORD and KEYED attributes. PL/I requires you to declare all UPDATE files with the DIRECT attribute so that you can locate the individual records.

A KEYED file is simply a fixed-length record file. The key is the relative position of the record in the file based upon the fixed record size. You must use keys to access a KEYED file. PL/I automatically gives KEYED files the RECORD attribute.

The LINESIZE attribute applies only to STREAM OUTPUT files, and defines the maximum input or output line length in characters.

The PAGESIZE attribute applies only to STREAM PRINT files, and defines the number of lines per page.

The TITLE(c) attribute defines the programmatic connection between an internal filename and an external device or a file in the operating system's file system. If you do not specify a TITLE attribute, the external filename defaults to the value of the file reference, with the filetype DAT. Otherwise, PL/I evaluates the character string c to produce either a device name,

$CON	System Console
$LST	System List Device
$RDR	System Reader Device
$PUN	System Punch Device

or a disk file in the form,

```
d:filename.typ;password
```

where the drive specification d, the filetype typ, and the password are optional. If you specify a password in the TITLE attribute the ENVIRONMENT attribute (see Section 10.1.2) defaults to ENVIRONMENT(Password(Read)).

Either the filename or filetype or both, can be $1 or $2. If you specify $1 then PL/I takes the first default name from the command line and fills it into that position of the title. Similarly, $2 is taken from the second default name and filled into the position where it occurs.

You must specify the filename, and you cannot use an ambiguous, or wildcard, reference in the filename, filetype, or the drive specification. Also, you can open the physical I/O devices $CON, $RDR, $PUN, and $LST only as STREAM files. $RDR must have the INPUT attribute, and $PUN and $LST must have the OUTPUT attribute.

The ENVIRONMENT attribute defines fixed- and variable-length record sizes for RECORD files, internal buffer sizes, the file open mode, and password protection level.

The general form of the ENVIRONMENT attribute is

```
ENVIRONMENT(option,...)
```

where option is one or more the following:

- Locked | L
- Readonly | R
- Shared | S
- Password[(level)] | P[(level)]
- Fixed(i) | F(i)
- Buff(b) | B(b)

You can specify options in the ENVIRONMENT attribute in any order with the exception that Fixed(i) must precede Buff(b).

Locked mode is the default mode for opening a file, and prevents other users from accessing the file while it is open. Readonly allows more than one user to open the file for Read/Only access. Shared, or unlocked, mode means that more than one user can open the file and access it. In Shared mode, you can apply the LOCK and UNLOCK built-in functions to individual records in the file. You can abbreviate each of the open modes with a single character. Thus, you can specify either Locked or L, Readonly or R, and Shared or S.

The option Password[(level)], defines the password protection level of the file. The valid protection levels are:

■ Read	R
■ Write	W
■ Delete	D

Read means that the password is required to read the file; this is the default mode. Write means that the file can be read but the password is required to write to the file. Delete means that the file can be read or written to , but the password is required to delete the file. You can abbreviate each of the protection levels with a single character. Thus, you can specify Read or R, Write or W, and Delete or D.

The option Buff(b) directs the I/O system to buffer b bytes of storage, where b is a FIXED BINARY expression that PL/I internally rounds to the next higher multiple of 128 bytes. In this form, the I/O system assumes that the file has variable-length records and therefore cannot have the KEYED attribute because the record size is not fixed.

The option Fixed(i) defines a file with fixed-length records containing i bytes each, where i is a FIXED BINARY expression that PL/I internally rounds to the next multiple of 128 bytes. If you use this option, you must also specify the KEYED attribute. When using this option, the default buffer size is i bytes, rounded to the next higher multiple of 128 bytes.

The options Fixed(i),Buff(b) defines a file containing fixed-length records of i bytes, rounded as above, with a buffer size of b bytes, again, rounded. You can specify a fixed-length record larger than the buffer size. When using these options, you must also specify the KEYED attribute.

10.2 Establishing File Attributes

When executing the OPEN statement, PL/I establishes the file attributes before associating the file with an external data set. If the OPEN statement does not specify a complete set of attributes, PL/I augments them with implied attributes. Table 10-1 shows the implied attributes for each specified attribute.

Table 10-1. PL/I Implied Attributes

Specified Attribute	*Implied Attribute(s)*
DIRECT	RECORD KEYED
KEYED	RECORD
PRINT	STREAM OUTPUT
SEQUENTIAL	RECORD
UPDATE	RECORD

Note: that the OPEN statement cannot contain conflicting attributes either explicitly or by default through the mechanisms that give the implied attribute.

Each type of I/O statement implicitly determines a specific set of file attributes. If you use the OPEN statement to explicitly specify the attributes, the attributes implied by the I/O statement cannot conflict with the attributes supplied in the OPEN statement. Table 10-2 summarizes the valid attributes for each of the I/O statements.

Table 10-2. Valid File Attributes for each I/O Statement

I/O Statement	*Valid Attributes*
GET FILE(f) LIST	STREAM INPUT
PUT FILE(f) LIST	STREAM OUTPUT
GET FILE(f) EDIT	STREAM INPUT
PUT FILE(f) EDIT	STREAM OUTPUT
READ FILE(f) INTO(v)	STREAM INPUT
READ FILE(f) INTO(x)	RECORD INPUT SEQUENTIAL
READ FILE(f) INTO(x) KEYTO(k)	RECORD INPUT SEQUENTIAL KEYED ENVIRONMENT(Fixed(i))
READ FILE(f) INTO(x) KEY(k)	RECORD INPUT DIRECT KEYED ENVIRONMENT(Fixed(i))
	RECORD UPDATE DIRECT KEYED ENVIRONMENT(Fixed(i))

Table 10-2. (continued)

I/O Statement	*Valid Attributes*
WRITE FILE(f) FROM(v)	STREAM OUTPUT
WRITE FILE(f) FROM(x)	RECORD OUTPUT SEQUENTIAL
WRITE FILE(f) FROM(x) KEYFROM(k)	RECORD OUTPUT DIRECT KEYED ENVIRONMENT(Fixed(i))
	RECORD UPDATE DIRECT KEYED ENVIRONMENT(Fixed(i))

Table 10-3 summarizes the valid attributes that can be associated with any file either through an explicit OPEN statement, or implicitly by an I/O access statement.

Table 10-3. PL/I Valid File Attributes

Type	*Attribute*
STREAM	INPUT ENVIRONMENT TITLE
STREAM	OUTPUT ENVIRONMENT TITLE LINESIZE
STREAM	PRINT ENVIRONMENT TITLE LINESIZE PAGESIZE
RECORD	INPUT SEQUENTIAL ENVIRONMENT TITLE
RECORD	OUTPUT SEQUENTIAL ENVIRONMENT TITLE
RECORD	INPUT SEQUENTIAL KEYED ENVIRONMENT TITLE
RECORD	OUTPUT SEQUENTIAL KEYED ENVIRONMENT TITLE

Table 10-3. (continued)

Type	*Attribute*
RECORD	INPUT DIRECT KEYED ENVIRONMENT TITLE
RECORD	OUTPUT DIRECT KEYED ENVIRONMENT TITLE
RECORD	UPDATE DIRECT KEYED ENVIRONMENT TITLE

Note: that once established, the set of attributes applies only to the current opening of the file. You can close the file and reopen it with a different set of attributes.

Some examples of the OPEN statement are shown below. In each case, there is a source statement with the default and augmented attributes shown below the statement. Each file is assumed to be declared as a file constant.

Statement:
```
open file(f1);
```

Attributes: STREAM INPUT ENVIRONMENT(Locked,Buff(128))
TITLE('f1.DAT') LINESIZE(80)

Statement:
```
open file(f2) print env(r);
```

Attributes: STREAM OUTPUT PRINT ENVIRONMENT(Readonly,Buff(128)
TITLE('f2.DAT') LINESIZE(80) PAGESIZE(60)

Statement:
```
open file(f3) sequential
     title('new.fil;John');
```

Attributes: RECORD INPUT SEQUENTIAL
ENVIRONMENT(Locked,Password(Read),Buff(128))
TITLE('new.fil;John')

Statement:
```
open title('a:'||c) file(f4)
     direct keyed env(s,f(2000));
```

Attributes: RECORD DIRECT INPUT KEYED
ENVIRONMENT(Shared,Fixed(2048),Buff(2048))
TITLE('a:'||c)

Statement:
```
open update keyed file(f5)
     env(locked,f(300),b(100));
```
Attributes: RECORD DIRECT UPDATE KEYED
ENVIRONMENT(Locked,Fixed(384),Buff(128))
TITLE ('f5.DAT')

Statement:
```
open file(f6) input direct
     title('d:accounts.new;topaz')
     env(shared,p(d),f(100),b(2000));
```

Attributes: RECORD DIRECT INPUT KEYED
ENVIRONMENT(Shared,Password(Delete),Fixed(128),Buff(2048))
TITLE('d:accounts.new;topaz')

In each of the preceding examples, PL/I allows integer expressions wherever a constant appears. Thus the statement

```
open file(f1) linesize(k+3) pagesize(n-4) env(b(x+128));
```

is a valid form of the OPEN statement.

10.3 The CLOSE Statement

The CLOSE statement dissociates the file from the external data set, clears and frees the internal buffers and permanently records the output files on the disk. The general form of the CLOSE statement is

CLOSE FILE(file_id);

where file_id is a file reference. You can subsequently reopen the file using the OPEN statement described above. If the file is not open, PL/I ignores the CLOSE statement.

10.4 The File Parameter Block

PL/I associates every file constant with a File Parameter Block (FPB). A FPB is a statically allocated segment of memory containing information about the file. A file variable has no corresponding FPB until you assign it a file constant. Each FPB contains the following information:

- the file title naming the external device or data set associated with the file

- the column position that the run-time system maintains to locate the next position to get or put data in a STREAM file
- current line count in STREAM OUTPUT files
- current page count for PRINT files
- current record position
- line size
- page size
- fixed record size
- internal buffer size
- File Descriptor containing one of the valid sets of file attributes as described above

While the file is open, the run-time system maintains an entry in a data structure called the Open List, which is allocated from the free storage area. Also, while the file is open, the FPB contains a pointer to the address of the operating system File Control Block (FCB).

10.5 I/O Conditions

During I/O processing, the run-time system can signal several conditions relating to the access of a particular file. The conditions are

- ENDFILE(file-reference)
- UNDEFINEDFILE(file-reference)
- KEY(file-reference)
- ENDPAGE(file-reference)

where file reference is an expression that reduces to a file constant. The file constant that results does not have to be open.

10.5.1 The ENDFILE Condition

The run-time system signals the ENDFILE condition whenever it reads an end-of-file character, CTRL-Z or ^Z, from a STREAM file, or it encounters the physical end-of-file in a RECORD file being processed in SEQUENTIAL mode. A read operation on a DIRECT file using a key beyond the end-of-file also signals the ENDFILE condition. Output operations that exceed the disk storage capacity signal the ERROR(14) condition.

10.5.2 The UNDEFINDFILE Condition

The run-time system signals the UNDEFINEDFILE condition whenever a program attempts to open a file for INPUT, and the file does not exist on the specified disk. The run-time system also signals the UNDEFINEDFILE condition if a program attempts to open a password-protected file without the correct password. Attempting to access a physical device such as $CON, $LST, $RDR, $PUN as a KEYED or UPDATE file, also signals this condition.

10.5.3 The KEY Condition

The run-time system signals the KEY condition when a program attempts to access a key value beyond the storage capacity of the disk.

10.5.4 The ENDPAGE Condition

The run-time system signals the ENDPAGE condition for PRINT files when the value of the current line reaches the PAGESIZE for the specified file. The current line always begins at zero, and the run-time system increases it by one for each line-feed that is sent to the file. If the file is initially opened with PAGESIZE(0), then the run-time system never signals the ENDPAGE condition. The run-time system resets the current line to one whenever:

- a form-feed is sent to the output file
- a PUT statement with a PAGE option is executed
- the default system action for ENDPAGE is performed

If the run-time system signals the ENDPAGE condition during execution of a SKIP option, the SKIP processing ends.

If an ON-unit intercepts the ENDPAGE condition, but does not execute a PUT statement with the PAGE option, then the current line is not reset to one. That is, until the program executes a PUT statement with the PAGE option, the run-time system continues to increment the current line, and does not signal the ENDPAGE condition. The current line counts up to 32767 and then begins again at 1.

10.5.5 Default I/O ON units

If an ON-unit receives control for the ENDFILE, UNDEFINEDFILE, or KEY conditions and returns to the point where the signal occurred, the run-time system terminates the current I/O operation, and passes control to the statement following the I/O statement that signaled the condition.

If there is no ON-unit enabled for the ENDFILE, UNDEFINEDFILE, or KEY condition, the default system action ends the program and outputs an appropriate error message.

If there is no ENDPAGE ON-unit enabled, the default system action inserts a formfeed into the output file, and continues processing.

10.6 I/O Condition BIFs

PL/I has several built-in functions for condition handling. They are:

- ONFILE
- ONKEY
- PAGENO
- LINENO

10.6.1 The ONFILE Function

The ONFILE function returns a CHARACTER VARYING string value of the internal filename involved in the last I/O operation that signaled a condition. With a conversion error, the ONFILE function produces the name of the file that is active at the time. If a signaled condition does not involve a file, then ONFILE returns a null string. The following code segment illustrates the use of ONFILE.

```
on error(1)
   begin;
   put list('Bad Data in file:',onfile());
   goto retry;
   end;
retry:
```

10.6.2 The ONKEY Function

The ONKEY function returns the value of the key involved in the I/O operation that signaled the KEY condition. ONKEY is valid only in the ON-body of the activated ON-unit. The following code segment illustrates the use of ONKEY.

```
on key(newfile)
   put skip list('bad key',onkey());
```

10.6.3 The PAGENO and LINEO Functions

The PAGENO and LINENO functions return the current page number and current line number for the PRINT file named as the parameter. When the ENDPAGE condition is signaled as the result of a PUT statement, the line number is one greater than the page size for the file.

10.7 Predefined Files

PL/I contains two predefined file constants called SYSIN and SYSPRINT. You do not have to declare these file constants unless you make an explicit file reference to them with an OPEN, GET, PUT, READ, or WRITE statement.

Otherwise, PL/I opens SYSIN with the default attributes:

STREAM INPUT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON')
LINESIZE(80)

and SYSIN becomes the console keyboard.

PL/I opens SYSPRINT with the default attributes:

STREAM PRINT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON')
LINESIZE(80) PAGESIZE(0)

and SYSPRINT becomes the console output display.

10.8 I/O Categories

PL/I supports two general categories of file access:

- STREAM I/O (sequential access only)
- RECORD I/O (sequential or random access)

10.8.1 STREAM I/O

A STREAM file is a sequence of ASCII characters separated by linemarks and pagemarks. When transmitting the data in a STREAM file to and from external devices, PL/I can format the data and perform conversion to other data types. Section 11 contains complete descriptions of the STREAM I/O statements.

10.8.2 RECORD I/O

In RECORD I/O, individual data items are called records, and they vary in size according to the data declaration. PL/I does not perform data conversion when transmitting data using the RECORD I/O statements (see Section 12), but simply transfers the internal representation of the data item.

End of Section 10

Section 11
Stream I/O

PL/I supports three forms of STREAM I/O:

- LIST-directed, transfers data items without format specifications.
- Line-directed, allows access to variable length character data in an unedited form. Note that PL/I provides line-directed STREAM I/O using READ and WRITE statements that might not be available in other implementations of PL/I.
- EDIT-directed, allows formatted access to character data items (see Section 11.3).

The following rules apply to all STREAM I/O:

- The column position for a file is initially 1.
- Each occurrence of a linemark or pagemark resets the column position to 1.
- If the input or output character is a special character, the column position advances by one.
- On output, if the column position exceeds the line size, the run-time system writes a linemark, increments the line number by one, and resets the column position to one.
- When the line number exceeds the page size, the run-time system signals the ENDPAGE condition. If no ENDPAGE On-unit is enabled, the run-time system writes a pagemark, and resets the column position and line number to one.

The following naming conventions appear throughout this section when describing the various STREAM I/O statements:

Table 11-1. Stream I/O Naming Conventions

Name	*Meaning*
file_id	is the file identifier.
nl	is a FIXED BINARY expression that defines the number of linemarks to skip on input, or the number of linemarks to write preceding the data item on output.

Table 11-1. (continued)

Name	*Meaning*

input-list	is a list of variables separated by commas, to which PL/I transmits the data items from the input stream. The input list determines the number and order of the variables assigned by the input data in the stream. In PL/I, the variables must be scalar values. You can include iterative DO-groups in the input-list but they require an extra set of parentheses. The DO header format is the same as the DO statement except that the REPEAT clause is not allowed. The general format is: (item-1,...,item-n DO iteration). For example, the following two sequences of code are equivalent:
	
	`do i = 1 to 10;`
	`   put list(A(i));`
	`end;`
	
	`put list((A(i) do i = 1 to 10));`

output-list	is a list of output items consisting of variables, constants, or expressions, separated by commas. The output list can also include iterative DO-groups.

11.1 LIST-directed I/O

The following constraints apply to the input stream for list-directed I/O:

- Data items in the stream can be arithmetic constants, character-string constants, or bit-string constants.
- Each data item must be followed by a separator, that consists of a series of blanks, a comma optionally surrounded by blanks, or an end-of-line character.
- PL/I treats embedded tabs, CTRL-I, as blanks.
- Character string data that actually contain blanks or commas must be enclosed in apostrophes. Otherwise, PL/I treats the blanks or commas as separators.

- A comma as the first nonblank character in the input line, or two consecutive commas optionally separated by one or more blanks indicate a null field in the input stream. The null field indicates that no data is to be transmitted to the associated data item in an input list. Thus, the value of the target data item remains unchanged.

11.1.1 The GET LIST Statement

The GET LIST statement reads data using list-directed STREAM I/O. The general form of the GET LIST statement is:

```
GET [FILE(file_id)] [SKIP[(nl)]] LIST(input-list);
```

You can specify the options FILE or SKIP in any order; LIST must appear last. If you do not specify the FILE option, PL/I assumes FILE(SYSIN). In a GET statement with the SKIP option, the run-time system ignores nl linemarks. If you do not specify nl with the SKIP option, then the run-time system ignores one linemark.

After transmission of all data items to the variables named in the input-list, the column position in the input stream remains at the character following the last data item read.

You can optionally enclose character strings in the input stream in apostrophes. If you do so, the run-time system does not transmit the enclosing apostrophes to the input variable. Likewise, for bit-string constants, the run-time system does not transmit the enclosing apostrophes and the trailing B to the input variable.

PL/I limits input strings to one line. Thus, string input from the console only requires the leading apostrophe when the string ends with a carriage return.

11.1.2 The PUT LIST Statement

The PUT LIST statement writes data using list-directed STREAM I/O. The general form of the PUT LIST statement is:

```
PUT [FILE(file_id)] [SKIP[(nl)]] [PAGE] LIST(output-list);
```

You can specify the options FILE, SKIP or PAGE in any order; LIST must appear last. If you do not specify the FILE option, PL/I assumes FILE(SYSPRINT).

If you do not specify nl with the SKIP option, then nl defaults to 1. If nl = 0, the run-time system does not write a linemark but resets the column position to 1. In either case using the SKIP option, the run-time system resets the column position to 1.

The PAGE option is valid only for PRINT files. Whenever the run-time system writes a pagemark, both the column position and line number are reset to 1.

When writing data items to a STREAM file, PL/I converts the items in the output-list to their character-string representation. The run-time system uses blanks to separate the data on the output line. If the data item is longer than the number of characters left on the output line, the run-time system writes the item at the beginning of the next line. If the length of the character string representation of the data item exceeds the line size, the run-time system writes the data item by itself on a single line that extends past the line size.

If the output transmission exceeds the page size, PL/I signals the ENDPAGE condition.

PL/I usually writes character strings enclosed in apostrophes. Each embedded apostrophe is written as an apostrophe pair,''. However, if the file has the PRINT attribute, the additional apostrophes are omitted. PL/I always writes bit-string data enclosed within apostrophes followed by the letter B.

11.2 Line-directed I/O

PL/I supports two forms of the READ and WRITE statement for processing variable-length ASCII records in a STREAM file. The two forms, called READ Varying and WRITE Varying, are not generally available in other implementations of PL/I. PL/I programs should avoid using these statements if compatibility is important.

11.2.1 The READ Varying Statement

The READ Varying statement reads variable length STREAM INPUT files. The general form of the READ Varying statement is

```
READ [FILE(file_id)] INTO(v);
```

where v is a CHARACTER VARYING string variable. If you do not specify the FILE option, PL/I assumes FILE(SYSIN).

READ Varying reads data from the input file until it reaches the maximum length of v, or it reads a line-feed character. READ Varying sets the length of v to the number of characters read, *including* the line-feed character.

Note: If you do not explicitly specify the file's attributes in an OPEN statement, the READ Varying statement causes an implicit OPEN statement with the resulting file attributes, STREAM and INPUT.

Given the declaration

```
declare
   F file,
   1 buffer,
     2 buffch character(254) varying;
```

the statement

```
read file(F) into(buffer);
```

produces RECORD data transmission because the target is a structure, not a CHARACTER VARYING variable. However, PL/I interprets the statement

```
read file(F) into(buffch);
```

as ASCII STREAM INPUT data transmission because the target is CHARACTER VARYING.

The READ Varying statement is differentiated from the READ statement only by the fact that the target variable has the attributes, CHARACTER VARYING.

11.2.2 The WRITE Varying Statement

The WRITE Varying statement writes variable length ASCII STREAM data. The general form of the WRITE Varying statement is

    WRITE [FILE(file_id)] FROM(v);

where v is a CHARACTER VARYING string variable. If you do not specify the file option, PL/I assumes file (SYSPRINT).

PL/I adds no additional control characters to the output string. If the application requires control characters, you must include them in the string. Recall that PL/I allows embedded control characters as a part of string constants, denoted by a preceding ^ in the string.

Note: If you do not explicitly specify the file's attributes in an OPEN statement, the WRITE Varying statement causes an inplicit OPEN statement with the resulting file attributes, STREAM and OUTPUT.

For example, given the previous declaration, PL/I interprets the statement

```
write file(F) from(buffer);
```

as RECORD data transmission. PL/I interprets

```
write file(F) from(buffch);
```

as a WRITE Varying statement, operating on an ASCII STREAM OUTPUT file, because the source variable has the VARYING attribute.

The WRITE Varying statement is differentiated from the WRITE statement only by the fact that the source variable has the attributes, CHARACTER VARYING.

11.3 EDIT-directed I/O

The input-list and the output-list for EDIT-directed I/O are analogous to those for LIST-directed I/O. However, EDIT-directed I/O uses a format list to specify how PL/I reads and writes the data.

11.3.1 The Format List

The format list is a list of format items, separated by commas. There are three types of format items:

- Data format items describing the data items to be read.
- Control format items specifying the placement of the data items in the stream.
- Remote format items referencing another format list.

The general form of a format list is

[n] fmt-item ...[,[n] fmt-item]

where n is an integer constant value in the range 1 to 254 that gives the repetition factor of the following fmt-item. If omitted, PL/I assumes a repetition factor of one. The fmt-item is either a data format item or a control format item.

An fmt-item can also be a remote format item. In PL/I, however, a remote format item must be the only format in the list, and cannot be preceded by a repetition factor.

11.3.2 Data Format Items

Data format items read or write numeric or character fields to or from an external STREAM data set. PL/I supports the following data format items.

The A[(w)] Format

This format reads or writes w characters of character string data. With GET EDIT, you must include w to be compatible with full PL/I. However, PL/I allows you to omit w with GET EDIT, and the A format reads the remainder of the current line up to, but not including the carriage return line-feed.

Input Value	Format	Input Result
byte	A(6)	'byte'
Napoleon	A(10)	'Napoleon'
string	A	'string'

With PUT EDIT, if you omit w, then the A format assumes w to be the length of the output string. If w is greater than the output string length, then the A format adds blanks on the right. If w is less than the output string length, the A format truncates the string in the rightmost positions.

Value	Format	Output Result
abcdef	A(6)	abcdef
abcdef	A(3)	abc
ƀ	A(4)	ƀƀƀƀ

The B[n][(w)] Format

This format reads or writes bit-string data. With GET EDIT, you must include w. n gives the number of bits to be used for each digit. If you omit n, the default is 1, so B is equivalent to B1 and only 0 and 1 can be in the input stream; otherwise PL/I raises the ERROR(1) condition. The valid digits for each value of n are as follows:

n	valid digits
1	0,1
2	0,1,2,3,
3	0,1,2,3,4,5,6,7
4	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Input Value	Format	Input Result
00101	B(5)	'00101'B
22	B2(2)	'1010'B
7C4	B4(3)	'011111000100'B

With PUT EDIT, the B format first converts the variable to a bit-string type, and then converts it to its character string representation. If you do not include w, the B format outputs the resulting character string. If you include w and it is longer than the character string, then the B format pads the string, with blanks, on the right. If the resulting character string is longer than w, the run-time system raises the ERROR(1) condition.

Value	Format	Output Result
'00'B	B	00
'1'B	B(4)	0001
'011101'B	B3(2)	35

The E(w[,d]) Format

This format reads and writes floating-point data. With GET EDIT, the E format converts the input characters to FLOAT BINARY values. w defines the field width, while d gives the number of digits to the right of the decimal point.

Input Value	Format	Input Result
b̸b̸b̸b̸	E(4)	0
2.9E7	E(5,3)	.29E+8
345678	E(6,2)	.345678E+4

With PUT EDIT, the E format converts the data item to FLOAT BINARY and represents it in scientific notation. w must be at least 7 more than d, because the output field appears as ±n.ddddE±eee, where ± represents sign positions; n is the leading digit, dddd represents the fractional part of length d, and E±eee represents the exponent field.

Value	Format	Output Result
0	E(11,3)	b̸0.000E+000
4.7E-10	E(11,3)	b̸4.700E-010
-30	E(15)	b̸-3.000000E+001

The F(w[,d]) Format

This format reads and writes fixed-point arithmetic data. w is the width, the number of characters in the field, and d is the number of characters to the right of the decimal point.

With GET EDIT, the F format reads as many characters as specified by w. If the character string contains a decimal point, then the decimal point determines the scale. Otherwise, d determines the scale. The F format ignores leading and trailing blanks. If the field contains only blank characters, the F format reads the value zero.

Input Value	Format	Input Result
b̸b̸0b̸b̸	F(5)	0
b̸-6b̸	F(4)	-6
13.09	F(5)	14

With PUT EDIT, the F format converts the data item to FIXED DECIMAL, and then uses d to specify the scale of the output value. If d is omitted, the scale is zero. The F format rounds the output value unless the variable has precision 15. The F format suppresses leading zeros except for one immediately to the left of the decimal point.

Value	Format	Output Result
`0`	`F(5,1)`	`b̸b̸0.0`
`-27`	`F(5,1)`	`-27.0`
`.39`	`F(6,2)`	`b̸b̸0.39`

11.3.3 Control Format Items

Control format items are used for line, page, and space placement. PL/I processes control format items as they are encountered in the format list, and ignores any items that remain after the input list or output list is exhausted. PL/I supports the following control format items.

COLUMN(nc)

This item moves the format pointer to column nc in the input or output data stream. With GET EDIT, COLUMN ignores those characters passed over by positioning the format pointer to column nc. If the current column position is less than nc, the format pointer moves to column position nc. If the current column position is greater than nc, the pointer first moves to the next line, and then moves to the new column position nc. If nc exceeds the rightmost position on the line, the format pointer moves to the first column of the new line. With GET EDIT, movement of the format pointer discards input characters.

With PUT EDIT, COLUMN writes blanks in the process of positioning to column nc. Also, if the current position is greater than nc, the run-time system outputs a linemark, then outputs blanks until it reaches column nc of the new line. If nc exceeds line size, the run-time system writes a linemark and sets the column position to 1.

LINE(ln)

This item applies only to PRINT files and specifies the line number of the next data item to be written. The constant ln must be greater than zero. If the current line number is equal to ln, LINE(ln) has no effect. If the current line number is less than ln, then the run-time system outputs linemarks until the current line number equals ln. PL/I raises the ENDPAGE condition if sufficient linemarks are issued to exceed the current page size.

PAGE

This item is used only with PRINT files and it causes the run-time system to write a pagemark, increment the page number by one, and set the line number and column position to 1.

SKIP[(nl)]

This item specifies the number of linemarks nl to be skipped or written. If omitted, nl defaults to 1. The run-time system sets the column position to 1.

With GET EDIT, nl is the number of linemarks to skip before moving to the next format item. The run-time system discards the first line, if the program executes a SKIP(1) as the first format item immediately following an explicit or implicit OPEN operation. SKIP(0) is undefined for input streams.

With PUT EDIT, nl is the number of linemarks to be written. If the page size is exceeded in the process of writing linemarks in a PRINT file, the run-time system raises the ENDPAGE condition and, upon return from the ON-unit, stops processing the SKIP operation.

X(sp)

This item advances the format pointer sp positions in the input or output data stream. With GET EDIT, sp is the number of characters to be advanced. The run-time system ignores linemarks, and continues the operation on the next line. With PUT EDIT, sp is the number of blanks to be written. If the end of the line is reached, the run-time system writes a linemark, and the blank fill operation continues on the next line.

11.3.4 Remote Format Items

The remote format item uses the format list of a FORMAT statement in place of the format item. The form of a remote format item is

R(format-label)

where the format label is the label constant preceding a FORMAT statement, in the scope of the remote format item. In PL/I only the remote format item can appear in the format list, with no preceding repetition factor, as shown in the following example:

```
put edit(A,B,C) (r(ELSEWHERE));
```

11.3.5 The FORMAT Statement

The FORMAT statement defines a remote format item, and has the general form,

format label: FORMAT(format-list);

where the format label is the label constant corresponding to the FORMAT, and the format list is a list of format items analogous to those described in the previous section. For example, the FORMAT statement

```
L1: format(A(5), F(6,2),skip(3),A(2));
```

is referenced as a remote format by the statement,

```
get edit (A,B,C) (R(L1));
```

11.3.6 The Picture Format Item

The Picture data format item is used on output to edit numeric data in fixed-point decimal form. The value resulting from such an edit is a character string whose form is determined by the numeric value and the Picture specification in the Picture format item.

The form of a Picture format item is

P'picspec'

where picspec is a character-string constant describing the Picture specification.

The Picture format item can appear in a PUT EDIT statement like any other data format item.

Picture Syntax

The character-string constant that describes the picspec must consist of one or more special characters as shown in Table 11-2.

Table 11-2. Picture Format Characters

Character	*Purpose*
$ + − S	static or drifting characters
* Z	conditional digit characters
9	digit character
V	decimal point position character
/ , . : B	insertion characters
CR DB	credit and debit characters

These characters must satisfy certain rules of syntax. Insertion characters can occur anywhere in a valid picspec, with the exception that they must not separate the characters of either Picture character pair, CR and DB.

If all insertion characters of a picspec are removed, the resulting string must be acceptable to the nondeterministic, finite-state machine recognizer illustrated in Figure 11-1. That is, it must be possible beginning with the START node to trace through this diagram to ACCEPT, where transitions across an edge are allowed if the edge is unlabeled, or if the edge is labeled by the next character in the picspec.

The following character string constants define valid picspecs:

```
'BB$***,***V.99BB'
'$----,999V.99BCR'
'99:99:99'
'**/**/**'
':BBB$SSSS,SSS.VSSBBB:'.
```

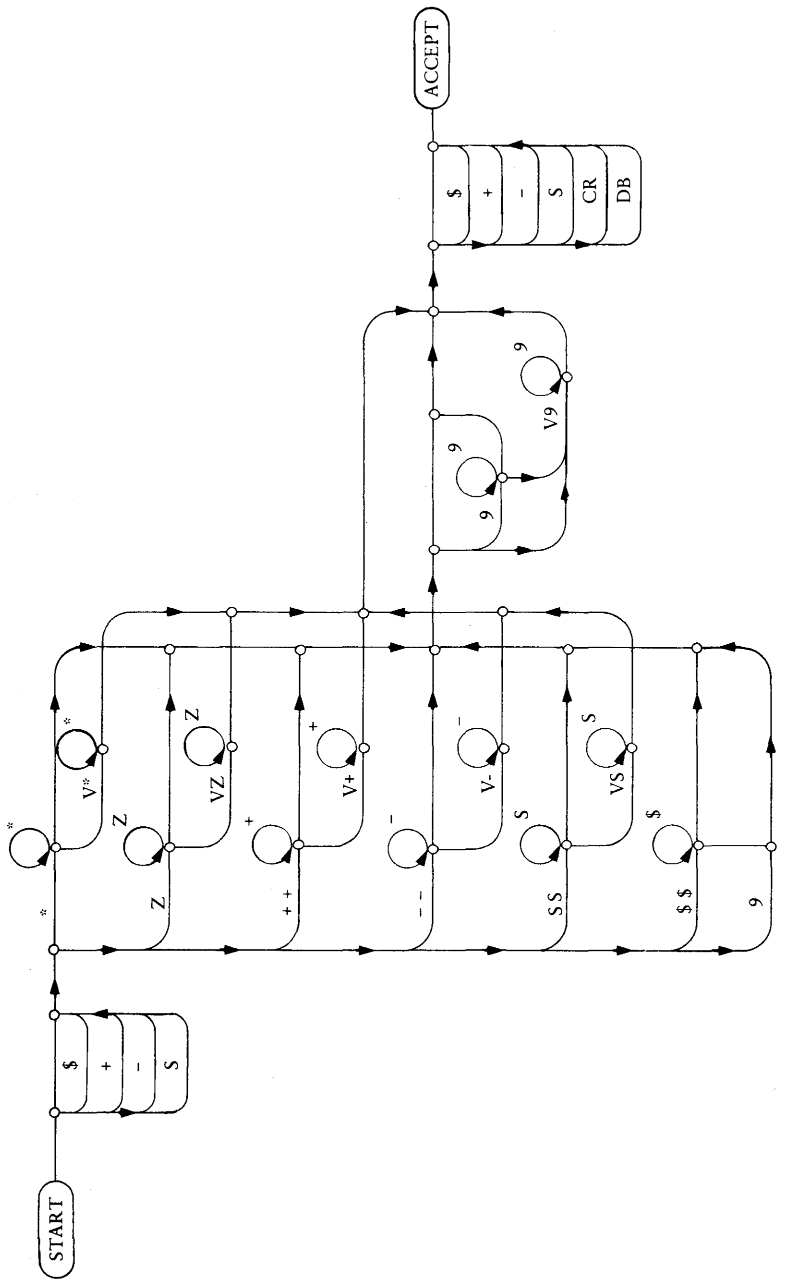


Figure 11-1. Picture Specification Recognizer

Picture Semantics

The types of Picture characters appearing in the specification determine how a picspec edits a numeric value into a character-string value.

In the picspec, certain characters occur as either static or drifting characters. These characters are:

- $: dollar sign
- + : plus sign
- – : minus sign
- S : upper-case S

Such a character is static if it appears only once in the picspec; otherwise, it is drifting. If it is drifting, all its occurrences except for one correspond to conditional digit positions.

In either case, these Picture characters, together with the sign of the numeric value, determine an output character, that occupies one position in the output. These output characters are shown in the following table.

Table 11-3. Picture Output Characters

Sign	*Static/Drifting Characters*			
	S	+	–	$
pos	+	+	' '	$
neg	–	' '	–	$

If the Picture character is static, the output character appears in the corresponding position of the output.

If the Picture character is drifting, then the output character appears exactly one position to the left of the first nonzero digit over which the Picture character drifts, or in the last position over which it drifts. All other occurrences of the drifting character are replaced by spaces, corresponding to the suppression of a zero digit in the numeric value.

The * and Z Characters

The characters * and Z are called conditional digit Picture characters or zero suppression characters. Each such character in the picspec is associated with a digit in the numeric value.

If the digit is a zero, the output character is an * or a blank. If the digit is nonzero, the output is the digit character.

The B, /, ., :, and , Characters

The Picture characters B, /, ., :, and , are called insertion characters. B is the space insertion character. The : is not an insertion character defined in the ANSI Standard, but is added in PL/I to display numeric data that represents time.

PL/I outputs insertion characters in the corresponding output position, unless the insertion character occurs in the field of a drifting character, or zero suppression character. If the insertion character occurs in the field of a drifting or zero suppression character that causes the suppression of numeric digits, then PL/I suppresses the insertion character following the preceding rules.

Note: in some PL/I implementations, B is an unconditional insertion character that always causes a space in the corresponding position of the output. According to the ANSI Standard, such a space in the output can be overwritten by a drifting character or, *, the zero suppression character.

The 9 Character

The Picture character 9 specifies that the corresponding digit in the numeric value occurs in the corresponding position of the output. Thus, 9 is an unconditional digit position.

The V Character

The V character establishes the correspondence between digits in the numeric value and the numeric digit positions in the picspec. This character only specifies the position where integral digits end and fractional digits begin. Thus the V character specifies the alignment of the picspec to the numeric value.

If you omit the V character, PL/I assumes that all the digit positions implied by the picspec refer to integral digit positions. Any fractional digits in the numeric value do not appear in the result.

Note: the V Picture character is the only character that does not correspond to a character position in the result. Thus the length of the resulting string equals the length of the picspec if V is omitted, but is one character less if V appears.

The V character also affects the suppression of characters. PL/I never suppresses fractional digits unless it suppresses all of the digits.

Beyond the V character PL/I turns OFF suppression if it is ON. As a result, PL/I does not suppress any insertion character occurring beyond the V Picture character, such as a decimal point, unless it suppresses everything.

The CR and DB Characters

The character pairs CR and DB, represent credit and debit. They act as sign characters. If either of them appear in the picspec, and if the sign of the numeric value is negative, then the specified pair occurs in the result. If the numeric value is positive, then the positions corresponding to these character pairs are replaced by two spaces.

Default Rules

If the numeric value is zero and if the picspec does not contain a 9 picture character, then the resulting output is all *s if the Picture character * occurs at all. Otherwise, the output is all spaces. This rule takes precedence over the other rules.

If the sign of the numeric value is negative, and if the picspec contains none of the sign Picture characters S, +, -, CR, or DB, then PL/I signals a conversion error, ERROR(1).

Each picspec implies a precision and scale for the numeric value in the result according to the following rules:

- Insertion characters and the character pairs CR and DB have no effect on precision and scale.

- The precision of the result equals one less than the number of static/drifting characters; or the number of zero suppression characters, plus the number of 9 characters.
- The scale of the result is zero if no V occurs.
- If V occurs, the scale of the result equals the number of drifting characters, or the number of zero suppression characters, or the number of 9 characters occurring after the V character.

Tables 11-4 and 11-5 illustrate some of the rules involving the use of Picture data format items.

Table 11-4. Picture Edited Output

Value	*picspec*	*Output Result*
0.00	BB$***,***V.99BB	$*******.00
0.01	BB$***,***V.99BB	$*******.01
0.25	BB$***,***V.99BB	$*******.25
1.50	BB$***,***V.99BB	$******1.50
12.34	BB$***,***V.99BB	$*****12.34
123.45	BB$***,***V.99BB	$****123.45
1234.56	BB$***,***V.99BB	$**1,234.56
12345.67	BB$***,***V.99BB	$*12,345.67
123456.78	BB$***,***V.99BB	$123,456.78
0.00	$$$$$B$$$V.99	$.00
0.01	$$$$$B$$$V.99	$.01
0.25	$$$$$B$$$V.99	$.25
1.50	$$$$$B$$$V.99	$1.50
12.34	$$$$$B$$$V.99	$12.34
123.45	$$$$$B$$$V.99	$123.45
1234.56	$$$$$B$$$V.99	$1 234.56
12345.67	$$$$$B$$$V.99	$12 345.67
123456.78	$$$$$B$$$V.99	$123 456.78

Table 11-4. (continued)

Value	*picspec*	*Output Result*
0.00	99/99/99	00/00/00
0.01	99/99/99	00/00/00
0.25	99/99/99	00/00/00
1.50	99/99/99	00/00/02
12.34	99/99/99	00/00/12
123.45	99/99/99	00/01/23
1234.56	99/99/99	00/12/35
12345.67	99/99/99	01/23/46
123456.78	99/99/99	12/34/57
0.00	**:**:**	********
0.01	**:**:**	********
0.25	**:**:**	********
1.50	**:**:**	*******2
12.34	**:**:**	******12
123.45	**:**:**	****1:23
1234.56	**:**:**	***12:35
12345.67	**:**:**	*1:23:46
123456.78	**:**:**	12:34:57
0.00	/++++,+++.V++/	
0.01	/++++,+++.V++/	/ +01/
0.25	/++++,+++.V++/	/ +25/
1.50	/++++,+++.V++/	/ +1.50/
12.34	/++++,+++.V++/	/ +12.34/
123.45	/++++,+++.V++/	/ +123.45/
1234.56	/++++,+++.V++/	/ +1,234.56/
12345.67	/++++,+++.V++/	/ +12,345.67/
123456.78	/++++,+++.V++/	/+123,456.78/

Table 11-5. Picture Edited Output

Value	*picspec*	*Output Result*
0.00	S***B***.V**	************
-0.01	S***B***.V**	-*********01
0.25	S***B***.V**	+*********25
-1.50	S***B***.V**	-*******1.50
12.34	S***B***.V**	+******12.34
-123.45	S***B***.V**	-*****123.45
1234.56	S***B***.V**	+**1 234.56
-12345.67	S***B***.V**	-*12 345.67
123456.78	S***B***.V**	+123 456.78
0.00	$SSSSBSSSV.SS	
-0.01	$SSSSBSSSV.SS	$ -.01
0.25	$SSSSBSSSV.SS	$ +.25
-1.50	$SSSSBSSSV.SS	$ -1.50
12.34	$SSSSBSSSV.SS	$ +12.34
-123.45	$SSSSBSSSV.SS	$ -123.45
1234.56	$SSSSBSSSV.SS	$ +1 234.56
-12345.67	$SSSSBSSSV.SS	$ -12 345.67
123456.78	$SSSSBSSSV.SS	$+123 456.78
0.00	***,***S	********
-0.01	***,***S	*******-
0.25	***,***S	*******+
-1.50	***,***S	******2-
12.34	***,***S	*****12+
-123.45	***,***S	****123-
1234.56	***,***S	**1,235+
-12345.67	***,***S	*12,346-
123456.78	***,***S	123,457+
0.00	$***,***V**CR	*************
-0.01	$***,***V**CR	$********01CR
0.25	$***,***V**CR	$********25
-1.50	$***,***V**CR	$*******150CR
12.34	$***,***V**CR	$******1234
-123.45	$***,***V**CR	$*****12345CR
1234.56	$***,***V**CR	$**1,23456
-12345.67	$***,***V**CR	$*12,34567CR
123456.78	$***,***V**CR	$123,45678

Table 11-5. (continued)

Value	*picspec*	*Output Result*
0.00	/++++,+++.V++/	
-0.01	/++++,+++.V++/	/ 01/
0.25	/++++,+++.V++/	/ +25/
-1.50	/++++,+++.V++/	/ 1.50/
12.34	/++++,+++.V++/	/ +12.34/
-123.45	/++++,+++.V++/	/ 123.45_
1234.56	/++++,+++.V++/	/ +1,234.56/
-12345.67	/++++,+++.V++/	/ 12,345.67/
123456.78	/++++,+++.V++/	/+123,456.78/

11.3.7 The GET EDIT Statement

The GET EDIT statement reads data using a format list. The general form of the GET EDIT statement is:

```
GET [FILE(file_id)] [SKIP[(nl)]]
      EDIT(input-list)(format-list);
```

You can specify the options FILE or SKIP, in any order. EDIT must appear last. If you do not specify the FILE option, PL/I assumes file(SYSIN).

The GET EDIT statement reads data items from the input stream into the variables given in the input-list until the input list is exhausted or the end-of-file is reached. The GET EDIT statement pairs each input list item with the next sequential format list item, applying control format items as they are encountered in the process. If the GET EDIT statement exhausts the input list before the end of the format list, remaining format items are ignored. If the GET EDIT statement exhausts the format list before the end of the input list, the format list is reprocessed from the beginning.

11.3.8 The PUT EDIT Statement

The PUT EDIT statement writes output data items according to a format list. The general form of the PUT EDIT statement is:

```
PUT [FILE(file_id)] [SKIP[(nl)]] [PAGE]
      EDIT(output-list)(format-list);
```

You can specify the options, FILE, SKIP, or PAGE, in any order. EDIT must appear last. If you do not specify the FILE option, PL/I assumes the file(SYSPRINT).

The PUT EDIT statement pairs output expressions from the output list with format items from the format list. The PUT EDIT statement also applies any control format items encountered during this process. The PUT EDIT statement ignores unprocessed format items at the end of the statement. If the PUT EDIT statement encounters the end of the output list during processing, the format list restarts from the beginning.

End of Section 11

Section 12
Record I/O

Record files contain binary data that PL/I transmits to or from an external device without conversion. There are two kinds of RECORD files:

- SEQUENTIAL, where PL/I accesses the records in the order they appear in the file.
- DIRECT, where PL/I randomly accesses the records through keys.

In the following discussion of RECORD I/O statements, file_id is a file variable or file constant; x is a scalar, or connected aggregate data type that does not have the attributes CHARACTER VARYING, and k is a FIXED BINARY key value or variable.

12.1 The READ Statement

The READ statement reads fixed or variable length RECORD files. The general form of the READ statement is:

```
READ FILE(file_id) INTO(x);
```

If you do not use the OPEN statement to open the file, the READ statement performs an implicit OPEN with the attributes RECORD, SEQUENTIAL, and INPUT.

The READ statement reads the number of bytes determined by the length of x. If you open the file with the ENVIRONMENT option specifying the size of the fixed-length record, the READ statement reads the amount of data according to the declared record size. If the length of x does not match the declared record size, the READ statement either pads x with zero-bits or truncates it on the right.

12.2 The READ with KEY Statement

The READ statement with the KEY option directly accesses individual records in a file. The form of the READ with KEY statement is

```
READ FILE(file_id) INTO(x) KEY(k);
```

where k is a FIXED BINARY expression that defines the relative record to access. Key values start at zero, and continue until the key value multiplied by the fixed-record length reaches the capacity of the disk.

If you do not use the OPEN statement to open the file, the READ with KEY statement performs an implicit OPEN with the attributes RECORD, INPUT, DIRECT, and KEYED. PL/I does not allow the READ with KEY statement to access variable length records.

12.3 The READ with KEYTO Statement

The READ statement with the KEYTO option extracts key values from an input file during sequential access. The program can save the key values in memory or in another file, and subsequently perform direct access on the records of the input file using the key values.

The general form of the READ with KEYTO statement is

```
READ FILE(file_id) INTO(x) KEYTO(k);
```

where k is a FIXED BINARY variable assigned to the relative record number of the record being read.

If you do not use the OPEN statement to open the file, the READ with KEYTO statement performs an implicit OPEN with the attributes RECORD, INPUT, SEQUENTIAL, and KEYED.

12.4 The WRITE Statement

The WRITE statement writes data from memory to the external data set without conversion. The general form of the WRITE statement is:

```
WRITE FILE(file_id) FROM(x);
```

If you do not use the OPEN statement to open the file, the WRITE statement performs an implicit open with the attributes RECORD, OUTPUT, and SEQUENTIAL.

The output record size is exactly the length of x. If you open the file with the ENVIRONMENT option specifying the fixed-length record size, the WRITE statement writes the amount of data according to the declared record size. If the length of x does not match the declared record size, the WRITE statement either pads x with zero-bits or truncates it on the right.

12.5 The WRITE with KEYFROM Statement

The WRITE with KEYFROM statement directly accesses a file for output. The general form is

```
WRITE FILE(file_id) FROM(x) KEYFROM(k);
```

where k denotes a FIXED BINARY expression yielding a key value that PL/I treats like the READ with KEY option shown above.

If you do not use the OPEN statement to open the file, the WRITE with KEYFROM statement performs an implicit OPEN with the attributes RECORD, DIRECT, OUTPUT, and KEYED.

End of Section 12

Section 13
Built-in Functions

A built-in function (BIF) is a computational subroutine provided as part of the PL/I Run-time Subroutine Library (RSL). You can use a BIF reference as a user-defined function reference.

You do not have to declare the name of a BIF. If you redeclare the name of a BIF in the program, you cannot reference it as a BIF within the scope of that declaration. However, you can use a BIF in a contained block by redeclaring the name with the attribute BUILTIN.

PL/I built-in functions are divided into the following categories:

- Arithmetic
- Mathemetical
- String Handling
- Conversion
- Condition Handling
- Miscellaneous

13.1 Arithmetic Functions

The arithmetic functions are:

ABS	FLOOR	MOD	TRUNC
CEIL	MAX	ROUND	
DIVIDE	MIN	SIGN	

The arithmetic BIFs return information about the attributes of specified arithmetic values, and perform common arithmetic calculations.

13.2 Mathematical Functions

The mathematical functions are:

ACOS	COS	LOG	SIND	TAND
ASIN	COSD	LOG2	SINH	TANH
ATAN	COSH	LOG10	SQRT	
ATAND	EXP	SIN	TAN	

The mathematical BIFs perform mathematical calculations in floating-point arithmetic. The mathematical functions include:

- the most commonly used trigonometric functions and their inverses
- base 2, base e, natural, and base 10, common, logarithmic functions
- the natural exponent function
- hyperbolic SIN and COS functions
- the square root function

Each of these functions accepts a single FLOAT BINARY argument and returns a FLOAT BINARY result. PL/I accepts other types of arguments but automatically converts them to FLOAT BINARY.

All of the function subroutines, with the exception of SQRT, use algorithms based on the Chebyshev polynomial approximations. The SQRT function subroutine is based on Newton's method.

Typically these algorithms scale the given argument into a finite interval, generally -1 <= X <= 1, and then evaluate the Chebyshev approximation using an appropriate recurrence relation. The greatest source of error in these routines results from the truncation of significant digits during the scaling process. Except for this, the subroutines have an average accuracy of 7.5 significant decimal digits for single-precision, 15 digits for double-precision.

13.3 String-handling Functions

The string-handling functions are:

- BOOL
- COLLATE
- INDEX
- LENGTH
- SUBSTR
- TRANSLATE
- VERIFY

The string-handling BIFs perform character-string and bit-string manipulation.

13.4 Conversion Functions

The conversion functions are:

- ASCII
- BINARY
- BIT
- CHARACTER
- DECIMAL
- DIVIDE
- FIXED
- FLOAT
- RANK
- UNSPEC

The conversion BIFs convert data from one type to another. PL/I uses these functions internally to perform automatic conversion.

13.5 Condition-handling Functions

The condition-handling functions are:

- ONCODE
- ONFILE
- ONKEY

The condition-handling BIFs return information about conditions signaled by the run-time system. These functions do not have parameters and return a value only when executed in an ON-unit. The ON-unit can be entered when the specified condition is programmatically signaled, or as the result of an interrupt caused by the occurrence of the specified condition.

13.6 Miscellaneous Functions

The miscellaneous BIFs are:

- ADDR
- DIMENSION
- HBOUND
- LBOUND

- LINENO
- LOCK
- NULL
- PAGENO
- UNLOCK

The miscellaneous BIFs return information about based variables, the current line number and page number of a file, information about array dimensions, and provide the ability to lock and unlock individual records within a file.

13.7 List of BIFs

The following sections describe the specific format, parameter attributes, purpose, and properties of each built-in function.

ABS

Category:	Arithmetic
Format:	ABS(X)
Parameters:	X can be any arithmetic expression.
Result:	Returns the absolute value of X.
Algorithm:	If X >= 0 then return X, otherwise return −X.
Result type:	Same as X.

ACOS

Category:	Mathematical
Format:	ACOS(X)
Parameter:	X is an arithmetic expression, −1 <= X <= 1.
Result:	Returns the arc cosine of X; for example, ACOS(X) is the angle in radians, whose cosine is X such that 0 <= ACOS(X) <= PI.

Result type:	FLOAT BINARY.
Algorithm:	ACOS(X) equals PI/2 – ASIN(X).
Error Condition:	
	If X is not in the interval −1 <= X <= 1 the run-time system signals the ERROR condition.

ADDR

Category:	Miscellaneous
Format:	ADDR(X)
Parameter:	X is a reference to a variable with connected storage.
Result:	Returns a pointer that identifies the storage location of the variable X.
Result type:	POINTER

ASCII

Category:	Conversion
Format:	ASCII(X)
Parameter:	X is a FIXED BINARY expression.
Result:	Returns a single character whose position in the ASCII collate sequence corresponds to X (see Appendix C for ASCII codes).
Result type:	CHARACTER(1)
Algorithm:	ASCII(X) equals SUBSTR(COLLATE(),MOD(X,128)+1,1).
Remark:	ASCII(X) is the inverse function of RANK(X).

<u>ASIN</u>

Category: Mathematical

Format: ASIN(X)

Parameter: X is an arithmetic expression, $-1 <= X <= 1$.

Result: Returns the arc sine of X; for example, ASIN(X) is the angle in radians, whose sine is X, such that $-PI/2 <= ASIN(X) <= PI/2$

Result type: FLOAT BINARY

Algorithm: Chebyshev polynomial approximation

Error Condition:

If X is not in the interval $-1 <= X <= 1$, the run-time system signals the ERROR condition.

<u>ATAN</u>

Category: Mathematical

Format: ATAN(X)

Parameter: X is any arithmetic expression.

Result: Returns the arc tangent of X; for example, ATAN(X) is the angle in radians, whose tangent is X, such that $-PI/2 <= ATAN(X) <= PI/2$

Result type: FLOAT BINARY

Algorithm: Chebyshev polynomial approximation

<u>ATAND</u>

Category: Mathematical

Format: ATAND(X)

Parameter:	X is any arithmetic expression.
Result:	Returns the arc tangent of X in degrees; for example, the angle, in degrees, whose tangent is X, such that $-90 <= \text{ATAND}(X) <= 90$
Result type:	FLOAT BINARY
Algorithm:	ATAND(X) equals 180/PI * ATAN(X)

<u>BINARY</u>

Category:	Conversion
Format:	BINARY(X[,P])
Parameter:	X is an arithmetic expression, or a string expression that can be converted to an arithmetic value. If X is DECIMAL with a nonzero scale factor, then P must be given, where P is an integer constant that specifies the precision of the result.
Result:	Returns a BINARY arithmetic value equivalent to X.
Result type:	If X is FLOAT BINARY, the result is FLOAT BINARY; otherwise it is FIXED BINARY.

<u>BIT</u>

Category:	Conversion
Format:	BIT(S[,L])
Parameter:	S is an arithmetic or string expression. L is a positive FIXED BINARY expression.
Result:	Converts S to a bit string of length L when L is specified. Otherwise it converts S to a bit string whose length is determined by the conversion rules in Section 4.3.3.
Result type:	BIT

BOOL

Category:	String
Format:	BOOL(X,Y,Z)
Parameters:	X is a bit expression. Y is a bit expression. Z is a bit-string constant, four-bits long.

Result: Returns a Boolean function on X and Y, specified by the bit-string constant Z as follows: Let Z1,Z2,Z3,Z4 be the bit values in Z, reading left to right. Then bit values A,B and the four-bit string Z determine the Boolean function BOOL(A,B,Z):

A	B	BOOL(A,B,Z)
0	0	Z1
0	1	Z2
1	0	Z3
1	1	Z4

This then induces the function BOOL(X,Y) on bit strings X and Y as follows. If X and Y do not have the same length, the shorter string is padded on the right with zero-bits until they have the same length. Then BOOL(X,Y,Z) is defined to be the bit string whose Nth bit is obtained from the preceding table by letting A be the Nth bit of X and B the Nth bit of Y.

Result type: BIT(n) where n equals MAX(LENGTH(X),LENGTH(Y)).

Examples: `BOOL('0011'B,'0101'B,'1001'B)` returns `'1001'B`.
`BOOL('01011'B,'11','1001')` returns `'01100'`.

CEIL

Category: Arithmetic

Format:	CEIL(X)
Parameter:	X is any arithmetic expression.
Result:	Returns the smallest integer >= to X.
Algorithm:	−FLOOR(−X)
Result type:	An integer value of the same type as X.

<u>CHARACTER</u>

Category:	Conversion
Format:	CHARACTER(S[,L])
Parameter:	S is an arithmetic or string expression, L is a positive FIXED BINARY expression.
Result:	S is converted to a character string of length L when L is specified; otherwise S is converted to a character string whose length is determined by the conversion rules of Section 4.
Result type:	CHARACTER

<u>COLLATE</u>

Category:	String
Format:	COLLATE()
Parameters:	None
Result:	Returns a character string of length 128 consisting of the set of characters in the ASCII character set in ascending order. (The ASCII character set is given in Appendix C.)
Result type:	CHARACTER(128)

COS

Category:	Mathematical
Format:	COS(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the cosine of X in radians.
Result type:	FLOAT BINARY
Algorithm:	Chebyshev polynomial approximation

COSD

Category:	Mathematical
Format:	COSD(X)
Parameter:	X is an arithmetic expression
Result:	Returns the cosine of X in degrees.
Result type:	FLOAT BINARY
Algorithm:	COSD(X) equals COS(X*PI/180)

COSH

Category:	Mathematical
Format:	COSH(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the hyperbolic cosine of X.
Result type:	FLOAT BINARY
Algorithm:	COSH(X) equals (EXP(X) + EXP(−X))/2

DECIMAL

Category:	Conversion
Format:	DECIMAL(X[,P[,Q]])
Parameter:	X is an arithmetic or string expression that can be converted to an arithmetic value. P is an integer constant, 1 <= P <= 15. Q is an integer constant, 0 <= Q <= P.
Result:	Converts X to a DECIMAL value. P and Q are optional but when specified represent the precision and scale factors, respectively. If only P is given, Q is assumed to be zero. If neither P nor Q is given, then the precision and scale factor of the result are determined by the rules for conversion given in Section 4.3.2.
Result type:	FIXED DECIMAL

DIMENSION

Category:	Miscellaneous
Format:	DIMENSION(X,N)
Parameters:	X is an array variable, N is a positive integer expression.
Result:	Returns a positive integer representing the extent of the Nth dimension of the array referenced by X.
Result type:	FIXED BINARY

DIVIDE

Category:	Arithmetic
Format:	DIVIDE(X,Y,P) or DIVIDE(X,Y,P,Q)
Parameters:	X and Y are arithmetic expressions.

Result:	Returns the quotient of X divided by Y, with the constants P, precision of the result, and Q, scale factor. Q assumed to be zero if not included. If X and Y are FIXED BINARY, Q must be omitted or equal to zero.
Result type:	The common arithmetic type of X and Y.

EXP

Category:	Mathematical
Format:	EXP(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the value of e to the power X, where e is the base of the natural logarithm.
Result type:	FLOAT BINARY
Algorithm:	Chebyshev polynomial approximation.

FIXED

Category:	Conversion
Format:	FIXED(X[,P[,Q]])
Parameters:	X is an arithmetic expression or string expression that can be converted to an arithmetic value. P is an integer constant. Q is an integer constant.
Result:	Converts X to a FIXED arithmetic value. P and Q are optional but when specified determine the precision and scale factor of the result. If only P is given, then Q is assumed to be zero. If neither P nor Q is given, then the precision and scale factor are determined by the conversion rules in Section 4.
Result type:	If X is FIXED DECIMAL or CHARACTER, the result is FIXED DECIMAL. Otherwise, it is FIXED BINARY.

FLOAT

Category:	Conversion
Format:	FLOAT(X[,P])
Parameter:	X is an arithmetic or string expression that can be converted to an arithmetic value. P is an optional positive integer constant.
Result:	Converts X to a FLOAT arithmetic value. P is optional but, when given, determines the precision of the result. If P is not given, the precision is determined by the conversion rules in Section 4.
Result type:	FLOAT BINARY

FLOOR

Category:	Arithmetic
Format:	FLOOR(X)
Parameter:	X is any arithmetic expression.
Result:	Computes the greatest integer <= X.
Result type:	An integer value of the same type as X.

HBOUND

Category:	Miscellaneous
Format:	HBOUND(X,N)
Parameters:	X is an array variable, N is a positive integer expression.
Result:	Returns the upper bound of the Nth dimension of the array variable X.
Result type:	FIXED BINARY

INDEX

Category:	String
Format:	INDEX(X,Y)
Parameters:	X and Y are string expressions of the same type, either bit or character.
Result:	Returns an integer value indicating the position of the leftmost occurrence of the string Y in the string X. If X or Y is null or if Y does not occur in X, INDEX returns the value zero.
Result type:	FIXED BINARY

LBOUND

Category:	Miscellaneous
Format:	LBOUND(X,N)
Parameters:	X is an array variable, N is a positive integer expression.
Result:	Returns the lower bound of the Nth dimension of the array referenced by X.
Result type:	FIXED BINARY

LENGTH

Category:	String
Format:	LENGTH(X)
Parameter:	X is a string expression, either bit or character.
Result:	Returns the number of characters or bits in the string X. If X has the attribute VARYING, LENGTH(X) returns the current length of X.
Result type:	FIXED BINARY

LINENO

Category:	Miscellaneous
Format:	LINENO(X)
Parameter:	X is a file value.
Result:	Returns the current line number of the file referenced by X. The file must have the PRINT attribute.
Result type:	FIXED BINARY

LOCK

Category:	Miscellaneous
Format:	LOCK(F,I)
Parameter:	F is a file constant or variable that must be opened in Shared mode. I is a FIXED BINARY(15) integer that gives the record number relative to the record size specified in the ENVIRONMENT option.
Result:	Returns a one-bit if the operation is successful or a zero-bit if unsuccessful. Locks the record specified by I so that no other user can lock or access it. The record remains locked until unlocked with the UNLOCK function, or the program terminates.
Result Type:	BIT(1)

LOG

Category:	Mathematical
Format:	LOG(X)
Parameter:	X is an arithmetic expression, $X > 0$.
Result:	Returns the natural logarithm of X.
Result type:	FLOAT BINARY

Algorithm:	Chebyshev polynomial approximation
Error Condition:	
	If X <= 0, the run-time system signals the ERROR condition.

LOG2

Category:	Mathematical
Format:	LOG2(X)
Parameter:	X is an arithmetic expression, X > 0.
Result:	Returns the logarithm of X to the base 2.
Result type:	FLOAT BINARY
Algorithm:	LOG2(X) equals LOG(X)/LOG(2)
Error Condition:	
	If X <= 0, the run-time system signals the ERROR condition.

LOG10

Category:	Mathematical
Format:	LOG10(X)
Parameter:	X is an arithmetic expression, X > 0.
Result:	Returns the logarithm of X to the base 10.
Result type:	FLOAT BINARY
Algorithm:	LOG10(X) equals LOG(X)/LOG(10)
Error Condition:	
	If X < 0, the run-time system signals the ERROR condition.

MAX

Category:	Arithmetic
Format:	MAX(X,Y)
Parameters:	X and Y are arithmetic expressions.
Result:	Returns the larger value of X and Y.
Algorithm:	If X >= Y then return X, otherwise return Y.
Result type:	The common arithmetic type of X and Y.

MIN

Category:	Arithmetic
Format:	MIN(X,Y)
Parameters:	X and Y are arithmetic expressions.
Result:	Returns the smaller value of X and Y.
Algorithm:	If X<= Y, then return X; otherwise return Y.
Result type:	The common arithmetic type of X and Y.

MOD

Category:	Arithmetic
Format:	MOD(X,Y)
Parameters:	X and Y are arithmetic expressions.
Result:	Returns the value X module Y.
Algorithm:	If Y=0 then return X, otherwise return X-(Y)*FLOOR(X/(Y)).
Result type:	The result is a value having the common arithmetic type of X and Y.

Examples:
```
MOD(7,3) returns 1
MOD(-7,3) returns 2
MOD(7,-3) returns 1
MOD(-7,-3) returns 2
```

Note: unless Y=0, MOD(X,Y) always returns a nonnegative value less than ABS(Y).

NULL

Category:	Miscellaneous
Format:	NULL[()]
Result:	Returns the null pointer value that points to an invalid storage location.
Result type:	POINTER

ONCODE

Category:	Condition
Format:	ONCODE()
Result:	Returns the value of the error subcode of the most recently signaled condition. The error conditions and their corresponding error numbers are listed in Section 9.4, Table 9-1.
Result type:	FIXED BINARY

ONFILE

Category:	Condition
Format:	ONFILE()
Result:	Returns the filename for which the most recent ENDFILE or ENDPAGE condition was signaled.
Result type:	CHARACTER

ONKEY

Category:	Condition
Format:	ONKEY()
Result:	Returns the character string value of the key for the record that signaled an input/output or conversion condition.
Result Type:	CHARACTER

PAGENO

Category:	Miscellaneous
Format:	PAGENO(X)
Parameter:	X is a file value.
Result:	Returns the page number of the file specified by X. The file must have the PRINT attribute.
Result type:	FIXED BINARY

RANK

Category:	Conversion
Format:	RANK(X)
Parameter:	X is a character value of length one.
Result:	Returns the integer representation of the ASCII character X (see Appendix C).
Result type:	FIXED BINARY
Algorithm:	RANK(X) equals INDEX(COLLATE(),X) -1

ROUND

Category:	Arithmetic
Format:	ROUND(X,K)
Parameters:	X is an arithmetic expression. K is a signed integer constant.
Result:	Returns X rounded to K digits to the right of the decimal point if K >= 0. Returns X rounded to -K digits to the left of the decimal point if K < 0.
Algorithm:	Return SIGN(X)*FLOOR(ABS(X)*B**N) + 0.5)/B**N where B=2 if X is BINARY B=10 if X is DECIMAL and N=K if X is FIXED else N=K-E if X is FLOAT and E is the exponent of X.
Result type:	Same as X

Examples:	`ROUND(12345.24689,3)` returns `12345.24700`
	`ROUND(34567,12345,-3)` returns `35000.00000`

SIGN

Category:	Arithmetic
Format:	SIGN(X)
Parameter:	X is any arithmetic expression.
Result:	Returns −1, 0, or 1 to indicate the sign of X.
Algorithm:	If X < 0 then return −1 If X = 0 then return 0 If X > 0 then return +1
Result type:	FIXED BINARY

SIN

Category:	Mathematical
Format:	SIN(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the sine of X in radians.
Result type:	FLOAT BINARY
Algorithm:	Chebyshev polynomial approximation

SIND

Category:	Mathematical
Format:	SIND(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the sine of X in degrees.
Result type:	FLOAT BINARY
Algorithm:	SIND(X) equals SIN(X*PI/180)

SINH

Category:	Mathematical
Format:	SINH(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the hyperbolic sine of X.
Result type:	FLOAT BINARY
Algorithm:	SINH(X) equals (EXP(X)-EXP(-X))/2

SQRT

Category:	Mathematical
Format:	SQRT(X)
Parameter:	X is an arithmetic expression, X >= 0.
Result:	Returns the square root of X.
Result type:	FLOAT BINARY
Algorithm:	Newton's method
Error Condition:	
	If X < 0, the run-time system signals the ERROR condition.

SUBSTR

Category:	String
Format:	SUBSTR(X,I[,J])
Parameters:	X is a string, either bit or character. I is a FIXED BINARY value. J is a FIXED BINARY value.
Result:	Returns a string that is a copy of the string X beginning at the Ith element and for a length J. If J is not given, it defaults to the length of the remainder of the string, equal to LENGTH(X) − I + 1.
Result type:	Same as X
Error Condition:	
	None. If the parameters are out of range, unpredictable results can occur.

TAN

Category:	Mathematical
Format:	TAN(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the tangent of X in radians.
Result type:	FLOAT BINARY
Algorithm:	TAN(X) = SIN(X)/COS(X)

Error Condition:

If COS(X) equals 0, then the run-time system signals the ERROR condition.

TAND

Category:	Mathematical
Format:	TAND(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the tangent of X in degrees.
Result type:	FLOAT BINARY
Algorithm:	TAND(X) = TAN(X*PI/180)

Error Condition:

If COS(X*PI/180) equals 0, the run-time system signals the ERROR condition.

TANH

Category:	Mathematical
Format:	TANH(X)
Parameter:	X is an arithmetic expression.
Result:	Returns the hyperbolic tangent of X.
Result type:	FLOAT BINARY
Algorithm:	TANH(X) = (EXP(X)-EXP(-X))/(EXP(X)+EXP(−X))

TRANSLATE

Category:	String
Format:	TRANSLATE(X,Y[,Z])
Parameters:	X is a character expression. Y is a character expression. Z is a character expression.
Result:	If Z does not occur, it is assumed to be COLLATE(). If Y is shorter than Z, it is padded to the right with blanks until its length equals the length of Z. Any occurrence of a character in Z in the string X is then replaced by the character in Y corresponding to that character in Z.
Result type:	Same as X
Examples:	`TRANSLATE('BDA','123','ABC')` returns `'2D1'`.

TRUNC

Category:	Arithmetic
Format:	TRUNC(X)
Parameter:	X is any arithmetic expression.

Result:	Returns the integer portion of X.
Algorithm:	If X < 0 then return (CEIL(X)) If X >= 0 then return (FLOOR(X))
Result type:	A signed integer value of the same type as X.

Examples:	`TRUNC(52.146)` returns `52`
	`TRUNC(-52.146)` returns `-52`

UNLOCK

Category:	Miscellaneous
Format:	UNLOCK(F,I)
Parameter:	F is a file constant or variable that must be opened in Shared mode. I is a FIXED BINARY(15) integer that gives the record number relative to the record size specified in the ENVIRONMENT option.
Result:	Returns a one-bit if the operation is successful or a zero-bit if unsuccessful. Unlocks the record specified by I so that other users can access it. The record remains unlocked until locked with the LOCK function, or the program terminates.
Result Type:	BIT(1)

UNSPEC

Category:	Miscellaneous
Format:	UNSPEC(X)
Parameter:	X is a reference to a data item whose internal representation in memory is 16 bits or less.
Result:	Returns the contents of the storage location occupied by X.
Result type:	A bit string whose length equals the length of the internal representation of the data item associated with X.

VERIFY

Category: String

Format: VERIFY(X,Y)

Parameters: X is a character expression.
Y is a character expression.

Result: Returns integer value 0 if each of the characters in X occurs in Y. Otherwise, returns an integer that indicates the position of the left-most character of X that does not occur in Y.

Result type: FIXED BINARY

Examples:

```
VERIFY(‘ABCDE’,‘ABDE’) returns 3
VERIFY(‘ABC123’,‘1A2B3C4D’) returns 0.
VERIFY(‘’,‘A’) returns 0.
VERIFY(‘A’,’’) returns 1.
```

End of Section 13

Appendix A
PL/I Statements

This appendix lists the PL/I statement formats in alphabetical order.

The ALLOCATE Statement

ALLOCATE based-variable SET(pointer-variable);

Example:

```
declare A character(16) based(P),
        P pointer;
allocate A set(P);
```

The ASSIGNMENT Statement

variable = expression;

Examples:

```
B = C*D;
unspec(E) = F(I);
```

The BEGIN Statement

begin;

The CALL Statement

CALL proc-name [(sub-1,...,sub-n)] [(arg-1, ...,arg-m)];

Example:

```
call P1;
call P2(A,B,C);
```

The CLOSE Statement

CLOSE FILE(file_id);

Examples:

```
close file(INP);
close file(OUT);
```

The DECLARE Statement

DECLARE|DCL [level] name [attribute-list]...

[,[level] name [attribute-list]];

Examples:

```
declare A fixed;
declare 1 B,
      2 C NAME character(20),
      2 D ADDRESS,
        3 STREET character(20),
        3 CITYST character(20),
        3 ZIP character(5);
declare ZZ(10) fixed;
declare A fixed external;
```

The DO Statement

DO [control-variable] do-specification;

where do-specification can be one of the following:

[start-exp [TO end-exp] [BY incr-exp]] [WHILE(condition)]
[start-exp [BY incr-exp] [TO end-exp]] [WHILE(condition)]
[start-exp [REPEAT repeat-exp]] [WHILE(condition)]

Examples:

```
do J=0;
do while(A<B);
do J = 1 TO 10;
do K = 10 TO 0 BY -2 while(A<B);
do P=START repeat P->NEXT while(P^=NULL);
```

The END Statement

END [label];

Examples:

```
end;
end P1;
```

The FORMAT Statement

label: FORMAT(format-list);

Examples:

```
L1: format(A(5));
L2: format(10 B4(2));
```

The FREE Statement

FREE [pointer-variable->] based-variable;

Examples:

```
free A;
free P->A;
```

The GET EDIT Statement

GET [FILE(file_id)] [SKIP[(nl)]]
EDIT(input-list) (format-list);

Examples:

```
get edit(A,B,C)((3)f(5,2));
get file(INP) edit((Z(I) do I = 1 to 3))(A);
```

The GET LIST Statement

GET [FILE(file_id)] [SKIP[(nl)]] LIST(input-list);

Examples:

```
get list(X,Y,Z);
```

The GOTO Statement

GOTO|GO TO label-constant|label-variable;

Examples:

```
go to the_end;
goto lab(K);
```

The IF Statement

IF condition THEN action-1 [ELSE [action-2]]

Examples:

```
if A=2 then B=A**2;
else;

if J>K then I = I+1;
else I = I+3;
```

The Null Statement

;

Examples:

```
;
else ;
```

The ON Statement

ON condition ON-unit,

Examples:

```
on endfile(INP)
  begin;
   put list('END OF INPUT');
   stop;
  end;

on error put list(oncode());
```

The OPEN Statement

OPEN FILE(file_id) [file-attributes];

Examples:

```
open file(INP) input;
open file(SYSPRINT) output;
```

The PROCEDURE Statement

proc-name: PROCEDURE|PROC [(parm-1, ...,parm-n)]
[OPTIONS(option,...)] [RETURNS(attribute-list)]
[RECURSIVE]

Examples:

```
P1:  proc(A,B,C);
P2:  procedure (ZZ) returns(float);
P3:  proc(N) returns(fixed bin) recursive;
P4:  procedure options(main);
```

The PUT EDIT Statement

PUT [FILE(file_id)] [SKIP[nl]] [PAGE]
EDIT(output-list)(format-list);

Examples:

```
put edit(A,B,C) (F(5,2),X(3),2E(10,2));
put edit((Z(I) do I = 1 to 10))(A);
```

The PUT LIST Statement

PUT [FILE(file_id)] [SKIP[(nl)]] [PAGE] LIST (output-list);

Examples:

```
put list(A,B,C);
put file(F) list((Z(I) do I = 1 to 10));
```

The READ Statement (for SEQUENTIAL RECORD files)

READ FILE(file_id) INTO(x);

Example:

```
read file(INP) into(XX);
```

The READ with KEY Statement

READ FILE(file_id) INTO(x) KEY(ikey);

Example:

```
read file(INP) into(STRUC) key(IKEY);
```

The READ with KEYTO Statement

READ FILE(file_id) INTO(x) KEYTO(keyto);

Example:

```
read file(INP) into(Z) keyto(IKEY);
```

The RETURN Statement

RETURN [(return-exp)];

Examples:

```
return;
return(X);
return(A**2);
```

The REVERT Statement

REVERT condition;

Examples:

```
revert error;
revert endfile;
```

The SIGNAL Statement

SIGNAL condition;

Examples:

```
signal error;
signal endfile(sysin);
```

The STOP Statement

STOP;

The WRITE Varying Statement (for STREAM files)

WRITE [FILE(file_id)] FROM(v);

Example:

```
declare (XX,YY) character(200) varying;
write file(OUTPUT) from(XX);
write from(YY);
```

The WRITE Statement (for SEQUENTIAL RECORD files)

WRITE FILE(file_id) FROM(x);

Examples:

```
write file(OUTP) from (XX);
write file(F) from(STRUC);
```

The WRITE with KEYFROM Statement

WRITE FILE(file_id) FROM(x) KEYFROM(ikey);

Example:

```
write file(KP) from(REC) keyfrom(IKEY);
```

End of Appendix A

Appendix B
Data Attributes

This appendix describes all the possible attributes with which program data can be associated in PL/I. Abbreviations of attributes are included. Refer to the relevant sections for full details of the attributes.

B.1 ALIGNED

ALIGNED is a data attribute that usually forces storage boundary alignment of a variable. It has no effect in PL/I but is included for compatibility with other implementations. For example:

```
declare A(0:3) bit (4) aligned;
```

B.2 AUTOMATIC | AUTO

AUTOMATIC is a storage class attribute that specifies that storage is allocated to the variable upon activation of the block containing the declaration. In PL/I, automatic storage is statically allocated, except for recursive procedures. For example:

```
declare A fixed binary; /* is equivalent to */

declare A fixed binary auto;
```

B.3 BASED or BASED(p) or BASED(q())

BASED is a storage class attribute that specifies user-controlled allocation for a variable. In this case, p is a pointer variable, and q is a pointer-valued function. For example:

```
declare A fixed binary based,
        B(5) character(10) based(p),
        C fixed binary based(f());
```

B.4 BINARY | BIN or BINARY (p) | BIN (p)

BINARY defines a BINARY variable with precision p.

for FIXED variables	p <= 15
for FLOAT variables	p <= 53

For example:

```
declare I fixed binary(7),
        F float binary(40);
```

B.5 BIT (n)

BIT (n) defines a bit string of length n, where n <= 16. For example:

```
declare A bit(3);
```

B.6 BUILTIN

BUILTIN specifies that the declared name is one of the PL/I built-in functions (BIFs). If you declare a BIF name in any block as a variable, then you must redeclare it with the BUILTIN attribute if you want to reference it as the BIF in any contained block. For example:

```
declare sqrt builtin;
```

B.7 CHARACTER(n) | CHAR(n)

CHARACTER (n) defines a character string of length n, where n <=254. For example:

```
declare A character(10),
        B(5) character(4);
```

B.8 DECIMAL[(p [,q])] | DEC[(p [,q])]

DECIMAL defines a decimal number with precision and scale (p,q), where p <= 15 and q <= p. If you do not specify q, the default is q = 0. If you do not specify either p or q, PL/I defaults to (7,0). For example:

```
declare A fixed decimal(6,2);
```

B.9 ENTRY[(att-1,att-2,...,att-n)]

ENTRY defines entry values, where att-1 to att-n is the attribute list of the parameters as given in the PROCEDURE definitions of the entry values. For example:

```
declare H entry,
        Z entry(10) (fixed),
        Y entry(float) returns(float),
        X entry variable;
```

B.10 ENVIRONMENT(options) | ENV(options)

ENVIRONMENT defines fixed- and variable-length record sizes for RECORD files, internal buffer sizes, the file open mode, and the password protection level. Options is one or more of the following:

Locked | L
Readonly | R
Shared | S
Password[(level)] | P[(level)]
Fixed(i) | F(i)
Buff(b) | B(b)

where i is the fixed-record length, and b is the internal buffer size. Both are expressed as integer constants. For example:

```
open file keyed env(f(100),b(4000));
open file(f6) input direct title('d:accounts.new;topaz')
              env(shared,password(d),f(100),b(2000));
```

B.11 EXTERNAL | EXT

EXTERNAL defines the scope of the declared item to be EXTERNAL. That is, the item is known in all blocks where it is declared as EXTERNAL. For example:

```
declare A character(8) external;
```

B.12 FILE

FILE defines file data. For example:

```
declare F file,
        FV file variable;
```

B.13 FIXED[(p [,q])]

FIXED defines fixed-point arithmetic data of precision and scale (p,q). If specified for BINARY data, q must be 0. For example:

```
declare A fixed binary,
        B fixed decimal(5,2);
```

B.14 FLOAT[(p)]

FLOAT defines floating-point arithmetic data of precision p, where p <= 53. For example:

```
declare A float binary;
```

B.15 INITIAL (value-list) | INIT (value-list)

INITIAL causes the Compiler to assign initial values to a STATIC variable before program execution. The value list is a list of constants, separated by commas, that can be converted to the variable type being initialized. Any constant in the list can be preceded by a repetition factor in parentheses. For example:

```
declare A character(3) static initial('ABC'),
        B(2) fixed binary static initial((2)5);
```

B.16 LABEL

LABEL defines a LABEL variable. For example:

```
declare somewhere label;
```

B.17 POINTER | PTR

POINTER defines a POINTER variable. For example:

```
declare (p,q) pointer;
```

B.18 RETURNS(attribute-list)

RETURNS (when used with the ENTRY attribute) describes the attribute list of the value returned by a function. For example:

```
declare A entry(float) returns(fixed);
```

B.19 STATIC

STATIC is a storage class attribute that causes the Compiler to allocate storage before program execution. For example:

```
declare A character(10) static,
        B fixed binary static initial(0);
```

B.20 VARIABLE

VARIABLE (when used with the FILE or ENTRY attributes) defines the item as a variable instead of a constant. For example:

```
declare F file variable,
        P entry variable;
```

B.21 VARYING | VAR

VARYING defines a varying length character string. For example:

```
declare A character(100) varying;
```

End of Appendix B

Appendix C
ASCII and Hexadecimal Conversions

ASCII stands for American Standard Code for Information Interchange. The code contains 96 printing and 32 nonprinting characters used to store data on a disk. Table C-1 defines ASCII symbols, and Table C-2 lists the ASCII and hexadecimal conversions. The table includes binary, decimal, hexadecimal, and ASCII conversions.

Table C-1. ASCII Symbols

Symbol	*Meaning*	*Symbol*	*Meaning*
ACK	acknowledge	FS	file separator
BEL	bell	GS	group separator
BS	backspace	HT	horizontal tabulation
CAN	cancel	LF	line-feed
CR	carriage return	NAK	negative acknowledge
DC	device control	NUL	null
DEL	delete	RS	record separator
DLE	data link escape	SI	shift in
EM	end of medium	SO	shift out
ENQ	enquiry	SOH	start of heading
EOT	end of transmission	SP	space
ESC	escape	STX	start of text
ETB	end of transmission	SUB	substitute
ETX	end of text	SYN	synchronous idle
FF	form-feed	US	unit separator
		VT	vertical tabulation

Table C-2. ASCII Conversion Table

Binary	*Decimal*	*Hexadecimal*	*ASCII*
0000000	0	0	NUL
0000001	1	1	SOH (CTRL-A)
0000010	2	2	STX (CTRL-B)
0000011	3	3	ETX (CTRL-C)

Table C-2. (continued)

Binary	*Decimal*	*Hexadecimal*	*ASCII*
0000100	4	4	EOT (CTRL-D)
0000101	5	5	ENQ (CTRL-E)
0000110	6	6	ACK (CTRL-F)
0000111	7	7	BEL (CTRL-G)
0001000	8	8	BS (CTRL-H)
0001001	9	9	HT (CTRL-I)
0001010	10	A	LF (CTRL-J)
0001011	11	B	VT (CTRL-K)
0001100	12	C	FF (CTRL-L)
0001101	13	D	CR (CTRL-M)
0001110	14	E	SO (CTRL-N)
0001111	15	F	SI (CTRL-O)
0010000	16	10	DLE (CTRL-P)
0010001	17	11	DC1 (CTRL-Q)
0010010	18	12	DC2 (CTRL-R)
0010011	19	13	DC3 (CTRL-S)
0010100	20	14	DC4 (CTRL-T)
0010101	21	15	NAK (CTRL-U)
0010110	22	16	SYN (CTRL-V)
0010111	23	17	ETB (CTRL-W)
0011000	24	18	CAN (CTRL-X)
0011001	25	19	EM (CTRL-Y)
0011010	26	1A	SUB (CTRL-Z)
0011011	27	1B	ESC (CTRL-[)
0011100	28	1C	FS (CTRL-\\)
0011101	29	1D	GS (CTRL-])
0011110	30	1E	RS (CTRL-^)
0011111	31	1F	US (CTRL-_)
0100000	32	20	(SPACE)
0100001	33	21	!
0100010	34	22	"
0100011	35	23	#
0100100	36	24	$
0100101	37	25	%
0100110	38	26	&
0100111	39	27	'
0101000	40	28	(
0101001	41	29	)

Table C-2. (continued)

Binary	*Decimal*	*Hexadecimal*	*ASCII*
0101010	42	2A	*
0101011	43	2B	+
0101100	44	2C	,
0101101	45	2D	-
0101110	46	2E	.
0101111	47	2F	/
0110000	48	30	0
0110001	49	31	1
0110010	50	32	2
0110011	51	33	3
0110100	52	34	4
0110101	53	35	5
0110110	54	36	6
0110111	55	37	7
0111000	56	38	8
0111001	57	39	9
0111010	58	3A	:
0111011	59	3B	;
0111100	60	3C	<
0111101	61	3D	=
0111110	62	3E	>
0111111	63	3F	?
1000000	64	40	@
1000001	65	41	A
1000010	66	42	B
1000011	67	43	C
1000100	68	44	D
1000101	69	45	E
1000110	70	46	F
1000111	71	47	G
1001000	72	48	H
1001001	73	49	I
1001010	74	4A	J
1001011	75	4B	K
1001100	76	4C	L
1001101	77	4D	M
1001110	78	4E	N
1001111	79	4F	O
1010000	80	50	P

Table C-2. (continued)

Binary	*Decimal*	*Hexadecimal*	*ASCII*
1010001	81	51	Q
1010010	82	52	R
1010011	83	53	S
1010100	84	54	T
1010101	85	55	U
1010110	86	56	V
1010111	87	57	W
1011000	88	58	X
1011001	89	59	Y
1011010	90	5A	Z
1011011	91	5B	[
1011100	92	5C	
1011101	93	5D	]
1011110	94	5E	^
1011111	95	5F	<
1100000	96	60	’
1100001	97	61	a
1100010	98	62	b
1100011	99	63	c
1100100	100	64	d
1100101	101	65	e
1100110	102	66	f
1100111	103	67	g
1101000	104	68	h
1101001	105	69	i
1101010	106	6A	j
1101011	107	6B	k
1101100	108	6C	l
1101101	109	6D	m
1101110	110	6E	n
1011111	111	6F	o
1110000	112	70	p
1110001	113	71	q
1110010	114	72	r
1110011	115	73	s
1110100	116	74	t
1110101	117	75	u
1110110	118	76	v

Table C-2. (continued)

Binary	*Decimal*	*Hexadecimal*	*ASCII*
1110111	119	77	w
1111000	120	78	x
1111001	121	79	y
1111010	122	7A	z
1111011	123	7B	{
1111100	124	7C	|
1111101	125	7D	}
1111110	126	7E	~
1111111	127	7F	DEL

End of Appendix C

Appendix D
Implementation Notes

The Digital Research implementation of PL/I is for microcomputers that use the 8080/8086, Z80, 8084/8088 or similar processors. It is formally based on the ANSI General Purpose Subset (Subset G) as specified by the ANSI PL/I Standardization Committee X3J1.

PL/I conforms to the Subset G specification with the following exceptions.

PL/I does not include the following attributes:

- DEFINED
- FLOAT DECIMAL
- PICTURE (it is implemented as an edit format item on output)
- Asterisk Extents and Dynamic Arrays

PL/I does not include the following built-in functions:

- ATANH
- DATE
- STRING
- TIME
- VALID

The following built-in functions are additions from the full PL/I:

- ASCII
- RANK

In PL/I, the %REPLACE statement is extended allowing multiple replaces in a single statement.

The following I/O facilities for ASCII file processing are added to PL/I:

- READ and WRITE statement forms for processing variable-length ASCII records
- The GET EDIT statement is extended to full record input in A format
- Control characters are allowed in string constants

PL/I is designed for use in limited resource environments. The following are implementation constraints imposed by the design.

- The PL/I condition stack is fixed at 16 levels. In any given block, PL/I stacks ON-units for the *same* condition. Therefore, you should not enable ON-units inside iterative loops because the condition stack can quickly overflow.
- An ON-unit cannot free storage for a variable that is being used when the condition is signaled, or close the file for which an I/O condition is signaled. The ON-unit must branch to a non-local label.
- PL/I does not support partially-subscripted, and/or partially-qualified mixed aggregate references that specify unconnected storage.
- PL/I does not support comparison operations for FIXED BINARY values whose sum or difference is greater than 32767 in absolute value.
- In the implementation of PL/I for the 8080 and Z80® processors, the Compiler produces relocatable object code in the MicroSoft® format. This format restricts the length of external names to six characters.
- In the implementation of PL/I for the 8086 and 8088 processors, the Compiler produces relocatable object code in the Intel® format. There are no restrictions on the length of external names with this format.

End of Appendix D

Appendix E
PL/I Bibliography

This appendix lists several PL/I programming reference books. Some are introductory textbooks for classroom use, while others are more advanced applications guides. Each reference is followed by a short description of the general content. You can obtain these books through your local bookstore, or order them directly from the publisher.

Although there are books now being prepared that specifically cover PL/I Subset G, the books listed below cover subsets such as PL/C and SP/k™ or the full IBM™ implementations of PL/I. The statement forms of PL/C and SP/k are generally included in the Subset G definition while full PL/I contains a number of language facilities excluded from the subset. Therefore, you should be aware that differences can arise even though the sample programs and definitions are substantially the same.

Your own reference library might consist of Lynch's book (12), that covers very general aspects of computing with introductory language details provided by the Xenakis book (14). Structured programming and program formulation is presented by one of the Conway books, such as (6). Additional application programming details are given in the Hughes book (9). Details of more advanced data structures are given in the Augenstein book (1).

Readers are encouraged to critique the individual books, and any additional reference material they find useful. Digital Research appreciates your comments and suggestions so that we can update this list.

(1) Augenstein, M., and A. Tenenbaum. *Data Structures and PL/I Programming*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979 (643p, Hardback, Typeset).

An advanced presentation of full PL/I. This is a college textbook presenting the PL/I language through a series of progressive examples covering recursion, list processing, trees and graphs, sorting, searching, hash coding, and storage management. An extensive bibliography is included. Emphasis is upon implementing data structures using a subset of full PL/I that nearly matches subset G. Structured programming is not emphasized.

(2) Bates, F., and M. Douglas. *Programming Language/One*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1970 (419p, Paperback, Hand Typed).

A simple introduction to PL/I. This book presents fundamental elements of full PL/I, with some emphasis on commercial processing including structures, records, formatting and error processing. Explanations are emphasized rather than examples. Structured programming is not emphasized.

(3) Cassel, D. *PL/I: A Structured Approach.* Reston Publishing, Inc., Reston, Virginia, 1978 (219p, Paperback, Typeset).

A middle level introduction to PL/I. A portion of full PL/I is presented emphasizing batch processing and commercial applications. Language elements are clearly presented, but there is no particular emphasis on program formulation or proper structuring, as the title implies.

(4) Clark, F. J. *Introduction to PL/I Programming.* Allyn and Bacon, Inc., Boston 1971 (243p, Paperback, Typeset).

A basic self-study introduction to PL/I through exercises. This text presents a portion of full PL/I from a traditional card-oriented approach, starting with a discussion of binary numbers and continuing through the basic statement types to simple STREAM and RECORD I/O. Structured programming is not emphasized, although commercial processing examples are given.

(5) Conway, R. *A Primer on Disciplined Programming.* Winthrop Publishers, Cambridge, Mass., 1978 (419p, Paperback, Computer Typed).

A textbook used for PL/C, Cornell University's dialect of PL/I. One of three college textbooks by Conway et. al., covering introductory programming, with emphasis on techniques used to formulate, develop, and test programs. Includes short discussions of searching and ordering lists, accounting, string operations, and interactive systems. Emphasis is upon structured programming practices and programming mechanisms rather than extensive examples of working programs.

(6) Conway, R., and D. Gries. *Primer on Structured Programming.* Winthrop Publishers, Cambridge, Mass., 1976 (397p, Paperback, Computer Typed).

A book on structured programming centered around PL/C. Essentially the same content as the previous book by Conway, with perhaps more emphasis on the operation of the PL/C programming system at Cornell.

(7) Conway, R., D. Gries, and D. Wortman. *Introduction to Structured Programming.* Winthrop Publishers, Cambridge, Mass., 1977 (420p, Paperback, Computer Typed).

A book on structured programming using Cornell's PL/C and Toronto's SP/k systems. Again, similar to Conway's first book with the addition of sections on file processing, and language translation using compilers and interpreters.

(8) Groner, G. *PL/I Programming in Technological Applications.* John Wiley & Sons, New York, 1971 (230p, Paperback, Typeset).

An introduction to engineering applications programming in PL/I. This book discusses full PL/I, with examples derived from batch processing under IBM implementations. Program formulation through flowcharting is presented, with many complete examples of scientific applications. Several examples of plot and graph generation are presented. Emphasis is upon explanations of FLOAT BINARY computations through complete examples. Programs are not particularly well structured.

(9) Hughes, J. K. *PL/I Structured Programming.* Second edition, John Wiley & Sons, New York, 1979 (825p, Hardback, Typeset).

A comprehensive guide to general PL/I programming. This is one of the more complete presentations of the full PL/I language. Topics include structured programming, processing simple data items, record and file handling, and list processing. Emphasis is toward commercial programming using IBM's PL/I.

(10) Hume, J. N. P., and R. C. Holt. *Structured Programming Using PL/I and SP/k.* Reston Publishing, Inc., Reston, Virginia 1975 (340p, Paperback, Computer Typed).

An introduction to structured PL/I programming. This textbook introduces PL/I through a graduated series of subsets called SP/1 through SP/8. Each successive subset incorporates more of the full PL/I language. The text begins with basic programming concepts, and progresses through the various PL/I language constructs. Sample programs include string and array handling, list processing, and file handling. Machine language, assembly language, and compiling is also presented. Emphasis is upon structured programming.

(11) Kennedy, M., and M. B. Solomon. *Structured PL/Zero Plus PL/One.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977 (695p, Paperback, Computer Typed).

A fairly comprehensive introduction to PL/I. This book covers the basic elements of PL/I in some detail, using PL/C for examples. IBM's PL/I Level F language is discussed briefly. Most language facilities are well illustrated in simple examples.

(12) Lynch, R. E., and J. R. Rice. *Computers, Their Impact and Use.* Holt, Rhinehart and Winston, New York, 1978 (440p, Paperback, Typeset).

A basic introductory book to computers and PL/I. This is a college textbook intended to introduce computers to nontechnical people. Half the book gives an overview of computers, their history, their impact upon society, and how they are used. Operating systems, languages, and language types are discussed. The remainder discusses IBM PL/I using a variety of applications, ranging up to simple file processing. Structured programming is not emphasized.

(13) Ruston, H. *Programming with PL/I.* McGraw Hill, New York, 1978 (541p, Paperback, Typeset).

A comprehensive textbook introduction to PL/I. This book presents PL/I from a batch processing viewpoint, using the full PL/I language for examples. Program construction through flowcharting is emphasized. Elements of PL/I are presented, including simple statements, control structures, arrays, strings, procedures, and file handling. Examples have a scientific orientation. Basics of error processing are discussed. Structured programming is not emphasized.

(14) Xenakis, J. J. *Structured PL/I Programming.* Duxbury Press, North Scituate, Mass., 1979 (413p, Paperback, Typeset).

A comprehensive introduction to PL/I, close to Subset G. Basic programming concepts are presented, with a brief history of programming languages. Elements of full PL/I are shown, including conversion between data types, arrays, strings, and procedures. A section on go-to-less programming is included, followed by a game-playing section that includes a tic-tac-toe program. The book is simple in scope and easy to read.

End of Appendix E

Appendix F
Glossary

aggregate: collection of related data items that you can reference together or individually.

algorithm: any procedure consisting of a finite number of unambiguous, repeatable steps that characterize the solution of a problem.

allocation: A) process of obtaining storage for a variable, or B) specific unit of storage that you obtain for a based variable.

argument: value that you pass to a subroutine or function.

argument list: zero or more arguments that you specify when invoking a procedure or a built-in function.

array: named collection of data items with the same attributes, and in which you access individual items, called elements, by subscripts.

ASCII character set: set of numeric values that represent characters and control information, established by American Standard Code for Information Interchange.

assignment statement: executable statement that assigns a value to a variable.

attribute: any characteristic of a data item, such as fixed- or floating-point, decimal or binary, extent, and so on.

automatic variable: variable for which the Compiler allocates storage when the block that declares it is activated. The storage is released when the block is deactivated.

based variable: variable that describes storage that you access using a pointer.

BEGIN block: one or more statements delimited by a BEGIN statement and a corresponding END statement. A BEGIN block is entered when control reaches the BEGIN statement. When control flows into a BEGIN block, PL/I creates a block activation for it and for the variables declared within it.

bit string: zero or more binary digits (0 or 1).

block: any sequence of PL/I statements delimited by one of the statement pairs PROCEDURE and END or BEGIN and END.

bound-pair: expression that sets the number of elements in each dimension of an array.

built-in function: any function provided as part of the PL/I language.

character string: zero or more ASCII characters.

comment: any sequence of characters appearing between the composite pairs /* and */. Comments provide documentary text and are ignored by the Compiler.

comparison operator: see **relational operator.**

Compiler: program that translates source statements of a high-level programming language into an object module. The object module consists of processor instructions and certain relocation information that the linkage editor uses to form a command (CMD) file.

computational: data type on which you can perform operations. The computational data types are arithmetic and string.

concatenation operator: operator, ||, that joins two string values to form a single string.

condition: any occurrence that interrupts the normal program execution and initiates a user-defined, or system default response.

condition name: PL/I keyword associated with a specific condition.

connected storage: contiguous storage locations.

constant: A) any literal value that you specify to represent a computational data item, or B) any entry or label name that you declare implicitly in context, or C) any identifier that you declare with one of the attributes ENTRY or FILE but without the VARIABLE attribute.

control variable: variable whose value changes on each iteration of a DO-group and that can be tested to determine whether or not to continue executing the statements in the DO-group.

conversion: process of transforming a value from one data type to another.

data type: class to which a data item belongs, and which determines the operations that you can perform on it.

declaration: explicit or implicit specification of an identifier and its data type.

dimension: set of bounds that determine one extent of an array.

DO-group: any sequence of executable statements delimited by a DO statement and a corresponding END statement.

element: any individual data item in an array, which you can reference with subscripts.

entry point: statement or instruction where the execution of a procedure begins.

expression: any valid combination of operands and operators that reduces to a single value.

extent: range between the low-bound and the high-bound for one dimension of an array.

external procedure: procedure that is not contained in any other procedure.

external variable: variable that is known in any block where you declare it with the EXTERNAL attribute.

file: A) in PL/I, the input source or output target that you specify in an I/O statement, or B) the collection of data on a mass storage device.

file constant: any identifier that you declare with the FILE attribute but not the VARIABLE attribute.

filetype: zero- to three-character component of a CP/M-86 file specification that generally describes the file's use.

FIXED BINARY: data type that represents integer values.

FIXED DECIMAL: data type that represents decimal values with a decimal point and a fixed number of fractional digits.

floating-point: data type that represents very small or very large numbers. A floating-point number has a mantissa and an optionally signed integer exponent.

flow of control: the sequence in which the processor executes the individual instructions in a program.

format item: value indicating data representation and formatting information used with EDIT-directed I/O.

format list: list of format items corresponding to data items for EDIT-directed I/O.

function: procedure that executes when you use its name in an expression, and that returns a value to its point of reference.

function reference: any reference to the name of a built-in function or a user-written function in a PL/I statement.

high bound: upper limit of an array dimension.

I/O category: general method you use to read or write data items in a file. The I/O categories are STREAM I/O and RECORD I/O.

identifier: name consisting of 1 to 31 characters that you specify for a variable, statement label, entry point, or file constant.

%INCLUDE file: external file from which the Compiler reads source text when compiling a PL/I program.

integer constant: any optionally signed string of decimal digits.

integer data: data represented as FIXED BINARY or FIXED DECIMAL with a zero scale factor.

internal procedure: procedure that is contained within some other procedure.

internal variable: variable whose value you can reference within the block that declares it and any blocks contained within the block that declares it.

iteration factor: integer constant enclosed in parentheses that specifies the number of times to use a value when initializing array elements, or the number of times to use a given format item in an EDIT-directed I/O statement.

key: (A) any value that you use to specify a particular record in a file, or (B) data item that is part of a record in an indexed sequential file, or (C) relative record number of a record in a RECORD file.

keyword: any PL/I identifier that has a specific meaning when you use it in the appropriate context.

label: any PL/I identifier, terminated by a colon, which you use to identify a statement.

level number: integer constant that defines the hierarchical relationship of a name within a structure with respect to other names in the structure.

library: file containing object modules and a directory of the external names within the object modules.

linker: program that arranges relocatable object modules into a command (CMD) file, and resolves references among external variables declared in the modules.

LIST-directed I/O: any transmission of data between a program and an external device, for which PL/I provides automatic data conversion and formatting.

listing: output file created by the Compiler that lists the statements in the source program, with corresponding line numbers, and additional information.

logical operator: operator that performs a logical operation on bit-string values.

low bound: lower limit of an array dimension.

main procedure: procedure that receives control when the program begins executing. The main procedure is always an external procedure.

major structure: name of an entire structure by which you can specify all members of the structure in a single reference. A major structure always has a level number of 1.

member: data item in a structure. A member can be a scalar data item, an array, or a structure.

memory: any addressable location that stores code or data.

minor structure: structure that is a member of a structure.

noncomputational: data item that is not string or arithmetic. The noncomputational data types are ENTRY, FILE, and LABEL.

nonlocal GOTO: GOTO statement that transfers program control to a statement in an encompassing block.

object module: output from the Compiler or assembler that you can link with other modules to form a command (CMD) file.

ON condition: any one of several named conditions that can interrupt a program and generate a signal.

ON-unit: PL/I statements specifying the action to take when a program signals a specific ON condition.

one-bit: the binary digit 1.

operator: symbol that directs PL/I to perform a specific function.

parameter: variable that PL/I matches with an argument when the program invokes a procedure.

parameter list: list of variable names whose values are determined when a procedure is invoked. The PROCEDURE statement for the procedure's entry point specifies the parameter list.

password: user-specified extension to a filename enabling file security.

Picture: character-string representation of an arithmetic value consisting of a character string constant defining the position of a decimal point, zero suppression, sign conventions.

pointer: data item whose value is the address of a storage location.

pointer-qualified reference: specification of a based variable in terms of a pointer value that indicates the location of the variable.

pointer qualifier: pointer reference and punctuation symbol that associates a specific storage location with a based variable.

precedence: priority of an operator that PL/I uses when evaluating operations in an expression. PL/I performs an operation with a higher precedence before an operation with a lower precedence.

precision: number of digits associated with an arithmetic data item.

prefix operator: operator that precedes a variable or constant to indicate or change its sign.

PRINT file: STREAM OUTPUT file for which PL/I aligns certain data on predefined tab stops, and controls the output with a specified page size and line size. In a PRINT file, PL/I does not enclose strings in apostrophes.

procedure: sequence of statements, delimited by a PROCEDURE statement and an END statement. A procedure can be a subroutine that you invoke with a CALL statement or a function that you invoke with a function reference.

procedure block: sequence of statements delimited by a PROCEDURE statement and an END statement. Control flows into a procedure block when you specify its name in a CALL statement or a function reference, at which point PL/I creates a block activation for it and for the internal variables declared within it.

pseudo-variable: name of a built-in function that you can use on the left-hand side of an assignment statement to give a special meaning to the assignment.

qualified reference: unambiguous reference to a member of a structure that specifies each higher-level name within the structure and separates the names with periods.

random access: I/O operation on a RECORD file where individual records within the file are accessed using FIXED BINARY values called keys.

record: organized collection of data that PL/I transmits using RECORD I/O statements.

RECORD file: file containing binary data that PL/I transmits without conversion.

RECORD I/O: transmission of data grouped in user-defined units called records.

recursive procedure: procedure that can invoke itself.

reference: appearance of an identifier in any context other than its declaration.

relational operator: operator that defines a relationship between two expressions and results in a Boolean value indicating whether the relationship is true or false.

return value: value returned by a function that replaces the function at its point of reference.

row-major order: order in which PL/I stores elements, or assigns values to elements in an array. In row-major order, the rightmost subscript varies the most rapidly.

Run-time Subroutine Library: library of procedures that support the execution of a PL/I program.

scalar: data item that is not an aggregate.

scale factor: number of fractional digits that you specify for a FIXED DECIMAL data item.

scope: set of blocks within a program in which the declaration of an identifier is known.

sequential access: access method that allows you to access records in a RECORD file serially.

sequential file: RECORD file in which the records are arranged serially. You can only add new records at the end of the file, and read records one after the other.

signal: mechanism by which PL/I indicates that a condition has occurred.

statement: valid sequence of PL/I keywords, identifiers, and special symbols that specifies an executable instruction or data declaration.

static variable: variable for which the Compiler allocates storage for the entire execution of a program.

storage: any region of memory that is associated with a particular variable.

storage class: attribute of a variable that describes how its storage is allocated and released by PL/I. The storage classes are AUTOMATIC, STATIC, and BASED.

STREAM I/O: transmission and interpretation of data in terms of sequences of ASCII characters delimited by spaces, tabs, commas, or fields defined by format items.

string data: data type consisting of either characters or bits.

structure: hierarchical arrangement of logically related data items, called members, that are not required to have the same data type.

structure reference: variable reference to an entire structure (as opposed to a member of a structure).

subroutine: procedure that receives control when you invoke it with a CALL statement.

subscript: integer expression specifying an individual element of an array or a label.

variable: data item whose value can change during the execution of a program.

variable reference: any reference to a variable including qualification by subscripts and member names.

zero-bit: the binary digit 0.

Note: Material in this appendix has been adapted in part from publication(s) of Digital Equipment Corporation™. The material so published herein is the sole responsibility of Digital Research Inc.

End of Appendix F

Index

Reader Comment Form

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date __________ Manual Title ____________________ Edition __________

1. What sections of this manual are especially helpful?

2. What suggestions do you have for improving this manual? What information is missing or incomplete? Where are examples needed?

3. Did you find errors in this manual? (Specify section and page number.)

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH™
P.O. Box 579
Pacific Grove, California
93950

Attn: Publication Production