PL/I-80™

LANGUAGE MANUAL

DIGITAL RESEARCH™

PL/I-80™ LANGUAGE MANUAL

PL/I-80 LANGUAGE MANUAL

Digital Research
P.O. Box 579
801 Lighthouse Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

The "PL/I-80 Language Manual" was prepared using the Digital Research TEX Text formatter.

```
*************************************
*  Second Printing: December 1980   *
*************************************
```

TABLE OF CONTENTS

INTRODUCTION

The purpose of the PL/I-80 Language Manual is to provide a relatively detailed but concise description of the PL/I-80 language for use by experienced PL/I programmers, and to act as a supplement to the accompanying Digital Research manual entitled "PL/I-80 Applications Guide." PL/I-80 is formally based upon the ANSI General Purpose Subset(Subset G) of PL/I as specified by the ANS PL/I Standardization Committee X3J1. The differences between PL/I-80 and the Subset G specification are as follows:

The following attributes are not included in PL/I-80:

DEFINED
FLOAT DECIMAL (FIXED DECIMAL is retained)
LIKE
PICTURE
FILE (allowed only in an OPEN statement in PL/I-80)
Asterisk Extents and Dynamic Arrays

The following builtin functions are not included in PL/I-80:

ATANH
DATE
STRING
TIME
VALID

The %REPLACE statement has been added to PL/I-80.

I/O facilities for ASCII file processing have been added:

READ and WRITE forms for variable length ASCII records
GET EDIT extended to full record input in A format
Control characters are allowed in string constants

The following builtin functions have been added:

ASCII
RANK

Throughout this document PL/I-80 statement formats will be described in their most general forms using the following notational conventions:

Words in capital letters represent PL/I-80 keywords.

Words in lower-case letters or in a combination of lower-case letters and digits separated by a hyphen will be described or defined more explicitly. These words represent variable information to be selected by the user.

Square brackets ([]) enclose options.

Ellipses (...) indicate that the immediately preceding item may occur once, or any number of times in succession.

Except for the above special characters, all other punctuation and special characters represent the actual occurrrence of those characters.

1. BASIC STRUCTURE

Basic structure consists of the low level organization of the source text of a PL/I-8Ø program. It includes, in addition to a specification of the character set of the language, a specification of the rules governing the structure of identifiers (both keywords and declared names), constants, delimiters, comments, and operators.

1.1. The character set.

The PL/I-8Ø character set consists of both upper and lower case letters, digits, and special symbols. The special symbols and a brief description of their use is given below:

=	equal or assignment
+	plus sign
-	minus sign
*	asterisk or multiply symbol
/	slash or divide symbol
(	left parenthesis
)	right parenthesis
,	comma
.	period
'	single quote or prime
%	percent symbol
;	semicolon
:	colon
^	logical "not" symbol
~	alternative "not" symbol
&	ampersand or logical "and" symbol
!	logical "or" symbol
\	alternative "or" symbol
|	alternative "or" symbol
>	greater than
<	less than
_	break or underscore
$	dollar sign
?	question mark

1.2. Identifiers.

An identifier is a string of from 1 to 31 characters which are either letters, digits, or the underscore, such that the first character is a letter. In PL/I-8Ø letters are always represented internally in upper case, therefore two identifiers which differ only in this respect determine the same identifier. PL/I-8Ø also allows the question mark character to be embedded within identifiers to allow access to external system entry points which often use this character. In general, embedded question marks should be avoided, however, to maintain upward compatibility with the full language.

Every identifier in the source text of a PL/I-80 program must be either a keyword or a declared name. Keywords are identifiers which have a special meaning in the PL/I-80 language, such as the names of built-in functions, statements, and data attributes (for a complete list of keywords, see the Appendixes and the "PL/I-80 Command Summary"). Declared names are identifiers whose use or meaning is defined by the programmer in a DECLARE statement (see Chapter 3). A keyword may also occur as a declared name, that is, appear in a declaration as a user defined identifier. In such a case the meaning of such an identifier in a PL/I-80 program will depend on how and where it appears. That is, the meaning is determined contextually.

1.3. Constants.

Constants are text items which, unlike some identifiers, have a fixed literal meaning which cannot change during the execution of a PL/I-80 program. The basic PL/I-80 constants are arithmetic constants, character string constants, and bit string constants. Arithmetic constants may be either FIXED BINARY, FLOAT BINARY, or FIXED DECIMAL. For a detailed description of the formats of each type of constant refer to the appropriate section of Chapter 3.

1.4. Delimiters and Separators.

In the source text of a PL/I-80 program it is essential that separate textual items, such as identifiers, be distinguishable. The textual items which can perform this role are called either delimiters or separators. Generally, delimiters enclose a textual item while separators separate textual items. In PL/I-80 each identifier and arithmetic constant must be preceded and followed by one or more delimiters or separators. Delimiters may be either spaces, operators, comments, or certain graphics delimiters.

Spaces: A space may be either a blank, a tab, or end of line character.

Operators: The four types of operators in PL/I-80 are:

Arithmetic operators:

+	addition or prefix plus
-	subtraction or prefix minus
*	multiplication
/	division
**	exponentiation

Comparison operators:

>	greater than
~>	not greater than
>=	greater than or equal to
=	equal to
~=	not equal to
<=	less than or equal to
<	less than
~<	not less than

Bit string operators:

~	not
&	and
! or |	or

The string operator:

!! or ||	concatenate

The above operators include so-called composite operators such as >=. These composite operators may not be separated by blanks.

Comments:

Comments are used to provide documentary statements in a program and have no effect on the execution of the program. A comment may be inserted wherever a delimiter is appropriate. The comment is initiated by the composite pair /* and is terminated by the reverse composite pair */.

Graphics delimiters and separators:

The following special characters may also function as delimiters or separators. Detailed descriptions of their use is given in an appropriate later section of this manual.

:	a colon is used as a separator for entry and label constants.
;	a semicolon is used to terminate statements.
,	a comma is used to separate elements of a list.
.	a period is used to separate items in a qualified name.
'	a quote is used as a delimiter for the specification of character and bit string constants.

->	the arrow is used as a separator in a pointer qualified reference.
=	an equal is used as a separator in an assignment statement.
(	left parenthesis
)	a right parenthesis together with a left parenthesis is used as a delimiter pair.

2. PROGRAM STRUCTURE

PL/I-8Ø is a free format block structured language. The basic program elements are statements which may be collected together into larger program elements called groups and blocks. In the following sections the rules governing the structure of these program elements are described.

2.1. PL/I-8Ø Statements.

With the exception of the assignment statement PL/I-8Ø statements consist of an optional label, followed by a keyword and statement body, and terminated by a semicolon. Statements fall into the following categories:

Structural statements which define blocks.

Declarative statements which describe data.

Executable statements which define action.

Null statements which indicate no operation.

Compound statements which are a collection of statements used to form a single statement (such as an IF statement).

Assignment statements.

Preprocessor statements which are compile-time instructions.

The specific structure of each type of statement is discussed in an appropriate section of this manual. A complete list of PL/I-8Ø statement formats and a brief description of their use is given in Appendix B.

2.2. Groups.

A group is a sequence of PL/I-8Ø statements that begins with a DO-statement and ends with an END statement, and may occur in one of two forms: the iterative DO and the non-iterative DO.

The non-iterative DO has the form:

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

The iterative DO has the form:

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

The following illustrates the use of groups:

```
first:
    do;         /* non-iterative do */
    j = x;
    if x > Ø then
        do;     /* non-iterative do */
        x = z;
        z = j * y;
        end;
    else
        x = y;
    end;
    do i = 1 to 1Ø; /* iterative do */
    a(i) = i*j;
    end;
```

A further discussion of the DO-statement occurs in Chapter 8.

2.3. Blocks.

A block is a sequence of statements delimited by either BEGIN and END statements (a BEGIN block), or by PROCEDURE and END statements (a PROCEDURE block). Blocks may be nested within one another, but are not allowed to overlap. Blocks are used to delimit the scope of declared names in a program. 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-8Ø statements constituting the block body. Note that the occurrence of the label option does not automatically add sufficient END statements to cause the block to balance, as found in some full-language implementations.

A PROCEDURE block has the following format:

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

where the label identifies the procedure, and Statement-1 through Statement-n are any PL/I-8Ø statements. Note that the label is optional for the END statement, but if included must match the label which names the procedure.

2.4. Scope of Names.

The scope of a variable refers in general to the extent of its definition in th e program. Variables may be either local, global, or EXTERNAL relative to a block in which they appear. There are two rules concerning the scope of data variables:

> The scope of a variable includes the block in which it is declared but not any block outside it. Anytime a variable is declared in a block, it becomes a local variable for that block.
>
> A variable is recognized in any block nested within the block in which it is declared. The variable is global to these nested block s. However, if the same name is declared in a subblock, a new variable is introduced in the subblock and becomes a local variable.

If neither of the above rules is satisfied, the variable is undefined. The program below illustrates the above rules:

```
pl:
    procedure;
    declare
        (a,b) fixed bin(7);
    a = 2;       /* a is local to pl */
    b = 3;       /* b is local to pl */
    p2:
         procedure;
         declare
             (c,b) fixed bin(7);
         b = 2;    /* b is local to p2 */
         c = a*b; /* c is local to p2 */
         a = a*b; /* a = 4 */
         end p2;
    put list (a,b);
    end pl;
```

This program produces the values 4 and 3. A new variable b is created in block P2 since it is a declared variable in that block. The PUT LIST statement is outside P2, therefore the value of the variable b of Pl is 3. Since there is no declaration for the identifier a in P2, the variable a referenced in P2 is the global variable a declared in Pl, and its value is changed by the assignment statement in P2. Note that the variable c declared in block P2 is unknown outside the procedure P 2.

Any variable declared as EXTERNAL is known to all blocks in which it is declared as EXTERNAL and in all contained blocks except where it is redeclared without the EXTERNAL attribute. For example,

```
p1:
    procedure;
    declare
        z fixed binary external;
        .
        .
        .
    p2:
        procedure;
        declare
            z fixed binary external;
            .
            .
            .
        p3:
            begin;
            declare
                z fixed binary;
                .
                .
                .
            end;
        end p2;
     end p1;
```

The variable z in P1 and P2 refer to the same external variable, but variable z in P3 is a local variable and is distinct from the external variable z. Note that due to the linkage editor format, all external names are truncated to the leftmost six characters and therefore long external names should be avoided to prevent conflicts.

2.5. Block Activation.

The essential difference between a BEGIN block and a PROCEDURE block is the manner in which they are activated and terminated. A BEGIN block is activated in the normal flow of statements in a program and is terminated when its corresponding END statement is encountered or when program control transfers outside the block.

A PROCEDURE block is activated only when invoked by a CALL statement or, if it is a function procedure, by a function reference in an expression. A procedure block is terminated when control is passed back to the point of call or reference. If a PROCEDURE block is encountered during the normal flow of execution, it is skipped.

2.6. Preprocessor Statements.

PL/I-8Ø allows source inclusion or modification at compile time through the use of preprocessor statements. Preprocessor statements are identified by a leading % symbol before the keyword

INCLUDE or REPLACE

The %INCLUDE statement copies PL/I-8Ø source text from an external CP/M file at compile time. The form of the statement is

```
%INCLUDE 'fname';
```

where fname is the name of a CP/M file to copy into the source program. If no drive name is given, the drive containing the source program is assumed. The copied text exactly replaces the %INCLUDE statement, and need not, itself, be a complete statement. Thus, for example, the %INCLUDE statement can be used to fill-out part of a structure declaration or format list. The program segment shown below provides an example:

```
F:
    PROC;
    DCL A FIXED;
    %INCLUDE 'STRUC.LIB';
    DCL C FLOAT;
    . . .
    END F;
```

directs the PL/I-8Ø compiler to include the source text from the file STRUC.LIB at the point of the %INCLUDE statement.

The %REPLACE statement allows textual replacement of constants for defined identifiers throughout the program. The form of the definition is:

```
%REPLACE rep-name BY constant-exp;
```

where rep-name is an identifier which, when subsequently encountered by the compiler, is replaced by the constant given by constant-exp. The constant-exp may be any string or arithmetic constant expression. Multiple %REPLACE statements can be written as a single %REPLACE statement, where the elements are separated by commas.

Note that the defined names in a %REPLACE statement do not obey the normal scoping rules. Hence, PL/I-8Ø requires that all %REPLACE statements occur at the outer block level, before any nested inner blocks. Generally, all %REPLACE statements are written directly following the procedure heading in order that they can be easily located. For example,

```
%REPLACE TRUE BY '1'B;
```

replaces all occurrences of the rep-name TRUE by the constant bit string '1'b, so that the statement

```
                    DO WHILE (TRUE);
```

is interpreted by the compiler as

```
                    DO WHILE ('1'B);
```

2.7. The Program.

As stated above, a procedure is a set of statements delimited by the PROCEDURE and END statements. A procedure not nested within another block is called an external procedure. A procedure contained entirely within an encompassing block is called an internal procedure. The source text of a PL/I-8Ø program may consist of one external procedure which may contain nested internal procedures or blocks. Each external procedure may be separately compiled and linked together to form a PL/I-8Ø object program. One of the external procedures forming the program must be the main procedure while the remaining procedures may be subroutines or function procedures. The format of a main procedure is:

```
label:
    PROCEDURE OPTIONS(MAIN);
        .
        .
        .
    Statements or Blocks
        .
        .
        .
    END [label];
```

For further details and examples of program structure and how they may be separately compiled, linked, and loaded, refer to the "PL/I-8Ø Applications Guide" and the "Link-8Ø Users Guide."

3. DATA ITEMS

The data items in a PL/I-80 program can be either constants or variables. A constant is a data item whose value cannot change during the execution of a program, while the value of a variable may change during execution. The data items in a program have certain additional properties, such as a range of subscript values, the operations which may be applied, or the amount of storage required. These properties, called attributes, are assigned to data variables in a DECLARE statement, or in some cases, such as constants, by system defaults. Data variables may represent single data items, or structures or arrays, referred to as data aggregates. A single data item, either a variable or constant, is called a scalar. In PL/I-80 there are essentially six forms of data: arithmetic, string, pointer, label, entry, and file data. In the following sections each of these data types are described in detail.

3.1. Arithmetic Data.

PL/I-80 supports three types of numeric data: FIXED BINARY, FLOAT BINARY, and FIXED DECIMAL. Each numeric 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 may contain and the scale q specifies the number of digits to the right of the decimal or binary point. The precision and scale of a variable can be explicitly specified when the variable is declared or implicitly given by default rules.

3.1.1. Fixed Binary.

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

$$1 <= p <= 15$$

Since this data type is internally represented in two's complement form, the range of a FIXED BINARY number is from -32768 to 32767. Storage allocated to a FIXED BINARY number depends on p. If p <= 7, then only one byte is allocated to the item. Otherwise two bytes are allocated. Default precision is (15). Assigning values to FIXED BINARY variables outside the legal range will produce undefined results.

A FIXED BINARY constant is written as a decimal integer. Such constants are considered FIXED BINARY data only if they appear in contexts which require fixed binary values, such as subscripts or arithmetic operations involving other FIXED BINARY data. Otherwise they default to FIXED DECIMAL. Conversion from other types of data normally occurs with truncation (See Chapter 7 for conversion rules). The value 1 is assigned to the variable I, for example, in the following program segment.

```
DECLARE I FIXED BINARY;
    I=1.99;
```

Declaring a variable as FIXED, BINARY, or FIXED BINARY is equivalent to declaring it as FIXED BINARY(15).

3.1.2. Fixed Decimal. Except for those numbers used in a FIXED BINARY context, all decimal constants with or without a decimal point default to FIXED DECIMAL. 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. T he range r of a FIXED DECIMAL number is:

```
-1Ø**(p-q) < r < 1Ø**(p-q)
```

where:

```
1 <= p <= 15 and

Ø <= q <= p
```

The default precision and scale is (7,Ø). Default precision and scale of decimal constants are determined by the the form of the constants themselves. For example:

```
3.25 defaults to (3,2)
3Ø2  defaults to (3,Ø)
```

The internal representation of a decimal number is packed BCD format. A value which has a scale greater than a FIXED DECIMAL variable will be truncated if assigned to the variable. Also if a value which has more significant digits to the left of the decimal point than are specified for the variable is assigned to the variable, then a FIXED OVERFLOW error occurs.

3.1.3. Float Binary. A FLOAT BINARY number has two parts, the fractional part (or mantissa) representing the significant digits of the number, and the exponential part indicating the scale factor. A variable declared as FLOAT BINARY (p) has a sign s, an integer exponent e, and p binary digits representing the fractional part of the number. The range of the magnitude of a FLOAT BINARY number r is approximately

5.88*10**-39 <= r <= 3.40*10**38

The range of p is:

1 <= p <= 24

The default precision of p is 24.

A FLOAT BINARY constant is written in scientific notation with a sequence of decimal digits with an optional decimal point followed by the upper or lower case letter E, followed by an optionally signed decimal integer exponent. For example:

A = 2.3E2;

assigns the value 230 to A. Declaring a variable FLOAT is equivalent to declaring it FLOAT BINARY.

3.1.4. Arithmetic Built-in Functions.

In addition to the arithmetic operators, PL/I-80 provides the following arithmetic built-in functions as part of the language:

ABS	ACOS	ASIN	ATAN	BINARY	CEIL	COS
COSD	COSH	DECIMAL	DIVIDE	EXP	FIXED	FLOAT
FLOOR	LOG	LOG10	LOG2	MAX	MIN	MOD
ROUND	SIGN	SIN	SIND	SINH	SQRT	TAN
TAND	TANH	TRUNC				

Detailed descriptions of the above built-in functions are given in Chapter 12.

3.2. String Data.

There are two types of string data in PL/I-80: character string data and bit string data. The value of a character string is a sequence of ASCII characters, including the empty or null sequence, and the value of a bit string is a non-empty sequence of bits. The length of a string is the number of characters or bits in the string. The rules governing the use of string data are described below.

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,

```
DECLARE A CHARACTER(10);
```

defines the variable A as a character string 10 characters long. If a character string assigned to A is shorter than A, blanks will be used to pad on the right up to the length of A. If a longer string is assigned to A, it will be truncated on the right.

Character string constants are written as a sequence of characters enclosed in quotes. If a quote mark is to be included in the string, it is represented as two consecutive quote marks. Thus the string constant whose value is What's Happening? is represented:

```
'What''s Happening?'
```

A null or empty character string is defined by using two consecutive quote marks. The null string has a length of zero.

Character string variables may also have the VARYING attribute which indicates that the variable may represent varying length strings up to a maximum length of n. To illustrate:

```
DECLARE A CHARACTER(10) VARYING;
```

defines A to represent any character string value whose length is not greater than 10.

PL/I-80 allows control characters to be included in string constants. In general, the up-arrow character "^" within a string constant indicates that a control character follows. The high order three bits of the character are masked to zero. Thus, the string '^M' or '^m' is converted to a carriage-return character. Similarly '^I' translates to the horizontal tab character. The occurrence of a double "^" within the string translates to a single '^' character. Note that this use of the up-arrow character is not generally available in the full language, and must be avoided if compatibility is required.

3.2.2. Bit String Data. A variable declared as BIT(n) is a bit string data item containing n binary digits, where n is a value between 1 and 16.

```
DECLARE A BIT(3);
```

defines a bit string of length 3. Assignments follow the same rules as for character strings, except that padding takes place with zeroes instead of blanks. Bit string variables may not be given the VARYING attribute.

Bit string constants are written as a sequence of digits Ø-9, and letters A-F enclosed in quote marks followed by the letter B and optionally followed by a digit in the range from 1 to 4 indicating the number of bits to be used to represent each digit in the sequence. That is, bit string constants may be written in base 2 (B or B1 format), base 4 (B2 format), base 8 (B3 format), or base 16 (B4 format). The characters or digits occurring in the sequence must be valid digits for the base specified by the format. The following examples illustrate the equivalence of the optional formats to the base 2 format:

```
'1Ø1'B1 is equivalent to '1Ø1'B
'1Ø1'B2 is equivalent to 'Ø1ØØØ1'B
'1Ø1'B3 is equivalent to 'ØØ1ØØØØØ1'B
'1Ø1'B4 is equivalent to 'ØØØ1ØØØØØØØ1'B
'9A'B4  is equivalent to '1ØØ11Ø1Ø'B
'77'B3  is equivalent to '111111'B
```

3.2.3. Concatenation. The infix operator || or !! can be used to concatenate either bit strings or character strings. Both operands must be of the same type, which is then the type of the result. 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 will have the VARYING attribute. In the following example

```
DECLARE
    A CHARACTER(3),
    B CHARACTER(6),
    C CHARACTER(2Ø) VARYING;
A = 'ABC';
B = 'ABCDEF';
C = A || B;
```

the character string 'ABCABCDEF' of length 9 is assigned to the variable C.

3.2.4. String Built-in Functions. The following string builtin functions are included in PL/I-8Ø and are described in detail in Chapter 12.

ASCII	BITS	BOOL	CHARACTER	COLLATE
INDEX	LENGTH	RANK	SUBSTR	TRANSLATE
UNSPEC	VERIFY			

3.3. Control Data Items.

Control data items, as opposed to problem data items, are used to control progr am flow. Identifiers used to label PL/I-8Ø statements and procedures are examples of control data items.

3.3.1. Label Data.

Label data consists of label constants and label variables. A label constant is a label identifier which 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 may be made to a label variable and follow the same rules as assignments for 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 in a DECLARE statement or implicitly by its use as a label constant. In particular, a transfer of control by the use of a GOTO statement is valid only when the target label is known in the block in which the GOTO statement occurs. Thus transfers of control using GOTO statements and labels are limited to the currently active block or a containing block.

Label constants may 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 which occur in its corresponding block. Label variables may be explicitly defined to be singly subscripted arrays in a DECLARE statement.

The only operators which may be used with label data are the equal and not equal comparison operators.

3.3.2. Entry Data.

Entry data items may be either entry constants or variables. The label of a PROCEDURE statement is called an entry constant. Entry constants may be external (identifying an entry point to an external procedure) or internal (identifying an entry point to a nested procedure). A variable declared as ENTRY VARIABLE is an entry variable which may be assigned entry constants or other entry variable values. As with label data, the only operators used with entry data are the equal and not equal comparison operators. Entry variables may also be subscripted. The following example shows how entry data items are used:

```
        DECLARE
            A ENTRY VARIABLE,
            (X,Y) FLOAT BINARY,
            F(3) ENTRY(FLOAT) RETURNS(FLOAT) VARIABLE,
            ZZ ENTRY(FLOAT) RETURNS(FLOAT);
        P1:
            PROC;
            X=5;
            END;
        P2:
            PROC;
            X=25;
            END;
        Y=9;
        IF Y = 5 THEN
            A = P1;
        ELSE
            A = P2;
        CALL A;
        F(2) = ZZ;
        Y = F(2)(X);
        PUT LIST(Y);
```

ENTRY data items are discussed further in Chapter 8.

3.4. Pointer Data.

Pointer data is used to address a location in memory. Its value is the address of a variable in the program. A pointer variable is declared in the following manner:

```
DECLARE X POINTER;
```

Pointer variables may be assigned other pointer variables, but as with label and entry data the only operators defined for pointer data are = and ~=. Certain types of variables must be qualified by pointer data when referencing them. The format of a pointer-qualified reference is:

```
pointer-variable -> based-variable
```

where pointer-variable points to the address of the based-variable. For example,

```
DECLARE P POINTER;
DECLARE X CHARACTER(2) BASED;
    .
    .
    .
P -> X = 'AA';
```

Pointer data items are also discussed in Chapter 6.

3.5. File Data.

File data items are used to represent information associated with an external device. PL/I-80 uses file constants and variables to access external data sets. A file constant is declared as follows:

```
DECLARE fname FILE;
```

where fname is a PL/I identifier assigned to represent the file name. In the case that fname is not a parameter, the file name is automatically taken as EXTERNAL so that it will access the same data set in all modules in which it is declared. Further, note that if no OPEN statement is executed with the TITLE option, the CP/M disk file named "fname.DAT" is accessed on the default drive.

A file variable is declared as follows:

```
DECLARE fname FILE VARIABLE;
```

FILE data is presented in more detail in Chapter 9, and in the manual entitled "PL/I-80 Applications Guide."

4. DATA AGGREGATES

In PL/I-8Ø data items may be grouped together to form arrays or structures. A variable that represents a group of data elements is either an array or a structure variable, and is referred to as a data aggregate.

4.1. Arrays.

An array is an ordered collection of data items whose elements have the same attributes. An entire array may be referenced by name, or an element of an array may be referenced through the use of integer subscripts. Subscripts denote the relative position of an element in an array. When an array is defined the following properties must be given: the data type of the elements, the dimension of the array, and the extent, or number of elements, in each dimension. The sum of the extents of each dimension of the array determines the total number of elements in the array. The dimension of an array is defined by the use of a dimension attribute list following the variable name in a DECLARE statement. A simple dimension attribute list consists of a list of positive integer constants, one for each dimension, specifying the extent of the subscript for that dimension. Each is separated by a comma, with the entire list enclosed in parentheses. For example,

```
DECLARE A(3,4) CHARACTER (2);
```

defines an array whose elements are character strings of length 2, whose dimension is two, such that the extent of the first dimension is 3, and the extent of the second dimension is 4. Thus A is an array with 3 rows and 4 columns whose entries are character strings of length two.

The range of values which the subscript corresponding to a particular dimension may assume is normally implied by the extent number for that dimension. In the array A, the row subscripts may range from 1 to 3, while the column subscripts may range from 1 to 4. The range of values may be explicitly defined for the subscripts of a particular dimension by replacing the extent value by a pair of integers in the form m:n, where m represents the lower bound, and n the upper bound for the subscripts for that dimension. The values m and n may be any integer values such that m is not greater than n. For example,

```
DECLARE B(-2:5,-5:5,5:1Ø) 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 1Ø, respectively. The corresponding extents are 8, 11, and 6 respectively. Thus B contains 25 data items of fixed binary type.

The elements of arrays are stored internally in row major order.

An element of an array is referenced by the name of the array followed by a list of subscript expressions separated by commas and enclosed in parentheses whose values specify the position of the element in the array. The subscript expressions must be of FIXED BINARY type and there must be one expression for each dimension of the array. The value of each subscript expression must be in the range defined for the subscript of that dimension.

An array variable may be assigned to another array variable directly, without the use of subscripts, if both array variables are defined with the same data type, dimension, and subscript ranges.

The DIMENSION, HBOUND, and LBOUND built-in functions are provided to access the extent, the upper bound, and the lower bound, respectively, of each dimension of an array. For detailed descriptions of these functions refer to Chapter 12.

4.2. Structures.

A structure is a hierarchically ordered set of data items. The data items contained in the structure are called its members and are not required to be the same type and may even be arrays or other structures (substructures). The main structure is called the major structure and any substructure is called a minor structure.

A structure is defined by specifying a name for the major structure, names and data attributes for its members, together with a level number for each name to define its level in the hierarchical order. 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 1. 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. Stucture names may not be given data type attributes, but may be given a dimension attribute (see below), and EXTERNAL, STATIC, or INITIAL attributes. For example, the definition of a structure for a billing account may appear as follows:

```
declare 1 bill,
          2 name,
            3 last_nm  character (20),
            3 first_nm character (20),
            3 mid_init 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 snkbar fixed decimal (10,2),
3 misc   fixed decimal (10,2),
3 dues   fixed decimal (10,2);
```

Since the name of a member of a structure may 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, ambiguities can arise in referencing the members of structures. These ambiguities arise only in the case that the member names are within a common scope of definition. To resolve such ambiguities, qualified names are used 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 which determine a unique reference to the member name. For example, consider 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 is ambiguous. The qualified names B.C or B.D or BB.C or BB.D uniquely identify the structure elements. Note that the fully qualified names would be:

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

4.3. Arrays of Structures.

Just as structures may have arrays for its members, arrays of structures may also be defined. An array whose elements are a single type of structure is defined by giving the structure name a dimension attribute when the structure is defined. Similarly, minor structures may be given a dimension attribute. For example,

```
          DECLARE 1 STULIST (1ØØ),
                    2 STUNAME.,
                      3 LASTNM  CHARACTER (1Ø),
                      3 FIRSTNM CHARACTER (1Ø),
                      3 MID_IN  CHARACTER (1),
                    2 SSN       CHARACTER (9),
                    2 COUNTRY   CHARACTER (1Ø),
                    2 GRADES(5) CHARACTER (2);
```

defines an array of structures whose major structure name is STULIST. Each structure element of the array has the array GRADES as a member. To reference an entry in the array, a qualified name is used together with subscripts for the structure names which 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 must appear in correct order.

For example, a reference to the 3rd GRADE entry for the 61st entry of the array STULIST may occur in any of the following forms:

```
                    STULIST(61).GRADE(3)
                    STULIST.GRADE(61,3)
                    STULIST(61,3).GRADE
```

5. DATA ATTRIBUTES AND THE DECLARE STATEMENT

Except for constant data and built-in names, every program data item must be explicitly defined through the use of the DECLARE statement and data attributes.

5.1. The Declare Statement.

All variable names in a program which are not the names of builtin functions or pseudo varaibles must be defined in a DECLARE statement. 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 properties associated to a name which characterize it as a data item are called the attributes of the name. Attributes can be defined implicitly by defaults or explicitly by their specification in a DECLARE statement. The general format of a DECLARE statement is:

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

where: name is the variable identifier, level is a positive integer used in a structure declaration as explained in the previous chapter, and the attribute-list describes the characteristics of the name. If the attribute-list is omitted, the attributes default to FIXED BINARY(15). Examples of DECLARE statements are:

```
DECLARE A CHARACTER (2) BASED,
        B FIXED BINARY INITIAL (0),
        C (100) FIXED DECIMAL (5,2);

DCL     1 BOOK,
          2 TITLE       CHAR(20),
          2 AUTHOR,
            3 LASTNM    CHAR(10),
            3 FIRSTNM   CHAR(10),
          2 PUBLISHER   CHAR(20);
```

Examples of simple declarations for each data type and data aggregate are given in Chapter 3 and Chapter 4.

The scope of a declared name is the region of the program in which the name with its associated attributes is known. In general, the occurrence of a name in a DECLARE statement implicitly defines the scope of the declared name to be internal, that is, the block in which the declaration occurs. If a name has the EXTERNAL attribute then its scope includes every block in which it is declared with the EXTERNAL attribute. If a variable is declared with the EXTERNAL attribute, it must also have the STATIC attribute since the name is associated with a single generation of storage. A name cannot be declared more than once in the same block except as the name of a member of a structure in such a way that unambiguous references may be made for each occurrence of the name. A name may not be declared with conflicting attributes in any two blocks in which it has the EXTERNAL attribute.

In PL/I, for convenience and simplicity, alternate forms of the DECLARE statement are allowed. First, in general, any sequence of DECLARE statements of the form:

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

can be written 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. Moreover, attributes shared by several item definitions can be factored to the right. That is, a sequence of definitions of the form:

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

can be written in an equivalent factored form:

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

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

```
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;
```

In general, 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, and the level numbers for members of structures must precede the member name. Both attributes may be factored. Since a dimension list is on the right of the name to which it applies, it is factored to the right as above. Since 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,

```
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);
```

An attribute list may not contain conflicting attributes, such as two data types, or two storage class attributes, but may omit a complete set of attributes, that is, enough attributes to characterize the data item. In this case, attributes are supplied by the PL/I-8Ø compiler according to 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 is specified for FIXED BINARY, FIXED BINARY(15) is assumed.

If no precision is specified for FIXED DECIMAL, FIXED DECIMAL(7,Ø) is assumed.

If no precision for FLOAT BINARY is specfied then FLOAT BINARY(24) is assumed.

5.2. List of Data Attributes.

The following paragraphs give the possible attributes with which program data may be associated. Abbreviations of attributes are included. Full details of the attributes are discussed in their relevant sections.

ALIGNED is a data attribute which normally forces storage boundary alignment of a variable. It has no effect in PL/I-8Ø.

AUTOMATIC is a storage class attribute which specifies that storage is allocated to the variable upon activation of the block containing the declaration. In PL/I-8Ø, automatic storage is statically allocated, except for recursion.

BASED or BASED(p) or BASED(q()) is a storage class attribute which specifies that allocation for a variable is user-controlled. In this case, p is a pointer variable, and q is a pointer valued function.

BINARY or BINARY (p) and BIN or BIN (p) define a BINARY variable with precision p.

BIT (n) defines a bit string of length n.

BUILTIN specifies that the name being declared is one of the built-in functions of the language. If a builtin function name is redefined in any block, then in order to reference the builtin function in a contained block, the builtin function name must be redeclared with the above attribute in the contained block.

CHARACTER (n) and CHAR (n) define a character string of length n.

DECIMAL [(p [,q])] and DEC [(p [,q])] define a DECIMAL number with precision (p,q). If q is not specified it is assumed to be zero. The default precision is (7,Ø).

ENTRY [(att-1,att-2,...,att-n)] defines ENTRY values. In the above, att-1 to att-n is the attribute list of th e parameters as given in the PROCEDURE definitions of the entry values.

ENVIRONMENT (options) or ENV (options) define fixed and variable length RECORD files where options may be one of the following:

```
                    F(lrecl)
                    B(lbuff)
                F(lrecl),B(lbuff)
```

The expression lrecl is the record length of a fixed length record file, and lbuff is the system buffer size.

EXTERNAL or EXT define the scope of the declared item to be EXTERNAL. That is the item is known in all blocks which declared it as EXTERNAL.

FILE defines file data. File attributes are discussed in Chapter 1Ø.

FIXED [(p [,q])] defines fixed-point arithmetic data of precision (p,q).

FLOAT [(p)] defines floating point arithmetic data of precision p.

INITIAL (value-list) and INIT (value-list) causes initial values to be assigned to a STATIC variable prior to program execution. The value-list is a list of constants, separated by commas, which can be converted to the variable type being initialized. Any constant in the list may be preceded by a repetition factor in parentheses.

LABEL defines a LABEL variable, and POINTER defines a POINTER variable.

RETURNS (attribute-list)) is given with the ENTRY attribute to describe the attribute-list of the value returned by a function.

STATIC is a storage class attribute that causes storage to be allocated prior to program execution.

VARIABLE is used with the FILE or ENTRY attributes to define the item as a variable and not a FILE or ENTRY constant.

VARYING and VAR define varying length character strings.

6. STORAGE MANAGEMENT

PL/I allows control over the allocation of storage for data areas. Storage may be statically allocated at compile-time, or dynamically during program execution. Dynamically allocated storage may be subsequently released for re-use. Every variable in a program has a storage class attribute. The storage class determines how and when storage is allocated for a variable. PL/I-80 includes three different storage classes:

STATIC
AUTOMATIC
BASED

In order to improve internal addressing mechanisms, AUTOMATIC storage is treated in the same way as STATIC storage, except in procedures which are marked as RECURSIVE. Storage class attributes are properties of elements, arrays, and major structure variables. These attributes cannot be given to entry names, file names, or members of data aggregates.

6.1. The STATIC Attribute.

A variable declared with the STATIC attribute is allocated prior to execution of the main procedure. The variables belonging to this storage class may have their data values initialized. The STATIC attribute is illustrated in the following declarations:

```
DECLARE A FIXED STATIC;
DECLARE B(100) CHAR(2) STATIC;
DECLARE 1 C STATIC,
          2 D CHAR(10),
          2 E FIXED;
```

6.2. The INITIAL Attribute.

The INITIAL attribute is used to give initial constant values to data items upon storage allocation. This initialized data becomes a part of the program module which is loaded when the program begins execution. The form of the INITIAL attribute is:

```
INITIAL | INIT (value[,value] ...)
```

where the initializing value is of the form:

```
[(iteration-factor)] constant-exp
```

The iteration-factor is a literal constant repeat count, which duplicates the constant-exp value an integral number of times. The constant-exp must be a literal constant value which is compatible with the data type being initialized, consisting of an optionally signed arithmetic constant, string constant, or NULL pointer value. Array data items can be initialized with a single statement which, in this case, begins with the first element of the array, and continues in row major order until the end of the set of initialized constants, which must 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. Thus, blank padding occurs on the right when a shorter string is assigned to a longer string variable, for example. Finally, only STATIC variables can have the INITIAL attribute in order to be compatible with the full language. Examples of the INITIAL attribute are shown below.

```
DECLARE A FIXED BINARY STATIC INITIAL (Ø);
DECLARE B(8) CHARACTER(2) INITIAL ((8) 'AB') STATIC;
DECLARE
    1 FCB STATIC,
      2 FCBDRIVE FIXED(7) INITIAL(Ø),
      2 FCBNAME  CHAR (8) INITIAL('EMP'),
      2 FCBTYPE  CHAR (3) INITIAL('DAT'),
      2 FCBEXT   BIT(8)   INITIAL('ØØ'B4),
      2 FCBFILL(19) BIT(8);
```

6.3. The AUTOMATIC Attribute.

Normally, the AUTOMATIC attribute forces data storage allocation upon entry to the PROCEDURE or BEGIN block in which the variable appears. In PL/I-8Ø, AUTOMATIC storage is statically allocated to improve variable addressing and execution speed. One exception is in the presence of recursion, where the AUTOMATIC variables must use the dynamic storage mechanism to prevent data overwrite on recursive calls. Note that the default storage class attribute for a variable not declared STATIC or BASED is AUTOMATIC, unless otherwise stated.

6.4. The BASED Attribute.

A variable declared with the BASED attribute is given storage explicitly through the ALLOCATE statement. Whenever a BASED variable is allocated, a corresponding POINTER variable is set to the address of the allocated BASED variable. The format for the BASED attribute is:

```
BASED [(pointer-ref)]
```

where the pointer-ref is an unsubscripted POINTER variable, or a function call, with zero arguments, which returns a POINTER value.

If a variable is declared with the BASED attribute, but the pointer-ref is omitted, then each reference to the variable must include a pointer-qualifier as shown below:

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

where pointer-exp is a pointer-valued expression, in order to locate the storage for the variable reference. If the pointer-ref option is included, then it is implicitly used as the base whenever the variable is reference without a pointer-qualifier. In this case, the pointer-ref is re-evaluated at each occurrence of the unqualified variable. The pointer variable or pointer-valued function name given in the pointer-ref is taken from the scope of the BASED declaration, even if a more local declaration exists with the same pointer-ref name. pointer-qualified reference has the format: Examples of BASED variable declarations are given below.

```
DECLARE A CHAR (8) BASED;
DECLARE B POINTER BASED(Q);
DECLARE C FIXED BASED(P);
DECLARE D BIT(8) BASED(F());
```

6.5. The ALLOCATE Statement.

The ALLOCATE statement is used to allocate storage for BASED variables, and takes the form:

```
ALLOCATE based-variable SET (pointer-variable);
```

A segment of storage is obtained from the dynamic storage area, sufficiently large to hold the value of the based-variable. The ERROR condition is raised if a segment of the requested size is not available. The based-variable must be an unsubscripted variable reference, where the variable is declared in the scope of the ALLOCATE statement with the BASED attribute. The allocation address is stored into the pointer-variable named in the SET clause. It is important to note that storage allocated in this manner remains allocated until a corresponding FREE operation takes place, using the allocation address held by the pointer-variable as an operand. The following program segment illustrates the use of the BASED attribute with the ALLOCATE statement:

```
DECLARE
    (P, Q) POINTER,
    X CHARACTER(2) BASED,
    Y FIXED BINARY BASED(P);
    . . .
    ALLOCATE X SET (Q);
    ALLOCATE Y SET (P);
    . . .
    Q -> X = 'AB';
    Y = Y + 1;
```

6.6. The NULL Built-In Function.

The NULL Built-in function returns a pointer value which is a unique non-valid storage address. This unique address is useful in marking various pointer values as empty, and is especially used in the construction of linked lists to mark the end-of-list element. The two forms of the NULL function call are:

NULL and NULL()

Note that pointer values do not necessarily begin with a NULL value upon program start-up, unless the NULL value is used in an INITIAL option in the declaration for the variable. Use of BASED variables and the NULL function is described extensively in the "PL/I Applications Guide."

6.7. The ADDR Built-In Function.

The ADDR built-in function returns a pointer value which addresses the segment of memory occupied by the variable-name given as the argument, and takes the form:

ADDR(variable-name)

Note that the variable-name must have an assigned memory address, and cannot be a temporary result created through the application of functions and operators.

Use of BASED variables in conjunction with the ADDR built-in function allows storage sharing in PL/I-8Ø. In this case, the based variable is not explicitly given storage using the ALLOCATE statement, but instead acts as a template which overlays an existing variable. the pointer base for the based variable is set to the address of the existing variable using the ADDR function. Subsequent access to the based variable thus accesses the overlayed variable. The program segment which follows illustrates storage sharing. In this case, 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.

```
DCL
    I FIXED,
    P POINTER,
    A CHAR(8),
    B(8) BIT(8) BASED (P);
    P = ADDR(A);
    GET LIST(A);
    PUT EDIT ((B(I) DO I = 1 TO 8))
        (B4(2));
```

6.8. The FREE Statement.

A BASED variable remains allocated until released with the FREE statement. The format of the FREE statement is:

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

where the pointer-variable addresses an allocation of storage which must have been previously obtained from the dynamic storage area using the ALLOCATE statement. If the pointer-variable is not given in the FREE statement, then the based-variable must be declared with the pointer-ref option. In this case, the storage addressed by the pointer-ref is given back to the dynamic storage area. The runtime subroutines which maintain the dynamic storage area automatically coalesce contiguous storage segments as they are released through the FREE statement. An example of the FREE statement in a non-functional program segment follows.

```
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. ASSIGNMENTS AND EXPRESSIONS

Assignment statements are used to set variables to the values of expressions or constants. The assignment statement contains no distinctive keyword, and takes the general form:

```
variable = expression;
```

where variable is a scalar element, an array, a structure name, or a pseudo-variable. The composition of expressions is given in the sections which follow.

7.1. Expressions.

An expression is a combination of operands and operators which, when computed at run-time, produce a single value. Syntactic rules govern the arrangement of references, operators, and parentheses used in an expression. A reference may be a constant, a variable, or a function. An operator defines the computation to perform using the operands to which it is applied. Parentheses may be used to enclose various portions of the expression. The proper formulation of operands, operators, and parentheses is presented in the following sections.

7.1.2. Prefix Expressions. A prefix expression consists of a unary operator followed by an operand. The operand is evaluated first, and the operator is applied to the result. Two examples of prefix expressions are shown below.

~A, and -SQRT(B).

7.1.2. Infix Expressions. An infix expression consists of two operands separated by an operator. The operands, which themselves may be expressions, are evaluated first, then the operator is applied to the result. Note that the order of evaluation is not specified in PL/I, so that the compiler can choose the most effective evaluation sequence. Two examples of infix expressions are shown below.

```
A+B        C**2
```

7.1.3. Precedence of Operators. The order in which operators are applied in an unparenthesized expression or subexpression is governed by a set precedence of operators. The fixed order of precedence in PL/I is listed below, from highest to lowest precedence. Operators with equal precedence are listed on the same line.

Exponentiation	**
NOT	~ or ^
Prefix Operators	+, -
Multiplication, Division	*, /
Addition, Subtraction	+, -
Concatenation	||, !!, or \\
Relational Operators	=, ~=, <, ~<, >, ~>, <=, >=
AND	&
OR	|, !, or \

The precedence of operators effectively inserts balanced parenthesis pair around the highest precedence operators first, descending to lower precedence operators until the entire expression is properly parenthesized. When equal precedence operators occur at the same level, prefix operators and exponentiation are evaluated from right to left, with the remaining operators evaluated from left to right. For example, the unparenthesized expression

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

is treated by the compiler as

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

7.1.4. Relational Operators. Relational operators are used to compare non-computational values for equality and inequality. Computational values can also be compared according to the normal algebraic rules. If the operands differ in form, they will first be converted to a common type, as described in the following section, before the comparison is made. A bit string of length one with value '1'B is produced if the comparison result is true, while 'Ø'B results if the relation is false. Character strings are compared by extending the shorter operand on the right with blanks until it is the length of the longer operand. The comparison is made character by character from left to right using the ASCII collating sequence as defined in Appendix A. Bit strings are compared by extending the shorter string on the right with zero bits. Comparison is then made bit by bit from left to right with Ø considered less than 1.

7.1.5. Bit String Operators. The bit string operators include:

^ ~ | ! \ &

where the first two symbols denote the logical negate (NOT) operator, the next three denote logical OR, and the last symbol represents logical AND.

Bit string operations are performed on a bit by bit basis. The unary NOT operator reverses each bit value in the bit-string operand: each Ø bit is changed to a 1, and each 1 bit is changed to a Ø bit. The OR and AND operators require two bit string operands. If the operands are of unequal length, the shorter one is extended on the right with zero bits until it is equal in length to the other operand. The resulting string length is equal to the longer of the two operands. The OR and AND operators follow the usual rules of Boolean Algebra:

x	y	x|y
Ø	Ø	Ø
Ø	1	1
1	Ø	1
1	1	1

x	y	x&y
Ø	Ø	Ø
Ø	1	Ø
1	Ø	Ø
1	1	1

(additional boolean functions are easily constructed using the BOOL built-in function).

7.1.6. Exponentiation. Exponentiation is computed as a series of multiplications if the exponent is a non-negative literal constant. Otherwise, the operation is evaluated using the LOG and EXP transcendental functions. A number of special cases for exponentiation are considered:

If X=Ø and Y>Ø then X**Y = Ø
If X=Ø and Y<Ø the ERROR condition is raised.
If X~=Ø and Y=Ø then X**Y = 1.
If X<Ø and Y is not an integer, the ERROR condition is raised.

7.2. Arithmetic Conversions.

Conversion between various data types may be required during the evaluation of an expression. All conversions involve the source data, an intermediate result, and the target data. The intermediate result is computed from the source data according to rules described below. The target data format is then derived from the intermediate result.

7.2.1. Arithmetic to Arithmetic Conversions. The general rules for arithmetic operand conversions are:

> If one operand is FIXED and the other is FLOAT, the common type is FLOAT. A FIXED BINARY(p) is converted to FLOAT BINARY(p), while a FIXED DECIMAL (p,q) is truncated and converted to FLOAT BINARY(p'), p' = min(ceil(p*3.32),24).
>
> If one operand is FIXED BINARY and the other type is FIXED DECIMAL, the common type is FIXED BINARY. A FIXED DECIMAL (p,q) becomes FIXED BINARY(p).

After the conversions to the common type have been made:

> If the operands are FLOAT BINARY, then the result is FLOAT BINARY, and the precision of the result becomes the greater precision of the two operands.
>
> If the operands are FIXED DECIMAL, let(p,q) be the precision of the first operand and (r,s) be the precision of the second operand. Then for addition or subtraction the precision of the result is:
>
> (MIN(15,MAX(p-q,r-s)+MAX(q,s)+1),MAX(q,s))
>
> For multiplication the precision of the result is:
>
> (MIN(15,p+r+1),q+s)
>
> For division, the precision of the result is:
>
> (15,15-p+q-s)
>
> Care should be taken when dividing FIXED DECIMAL values. The DIVIDE function may be used to control the precision of the quotient.
>
> If the operands are FIXED BINARY let (p) be the precision of the first operand and (r) be the precision of the second operand. Then for addition or subtraction the precision of the result is:
>
> (MIN(15,MAX(p,r)+1))
>
> For multiplication the precision of the result is:
>
> (MIN(15,p+r+1)
>
> The DIVIDE built-in function must be used to be compatible with the full language, and a scale factor of zero must be given, producing an integral result.

Truncation occurs if the precision of the result is insufficient to hold the number. Truncation occurs on the right for FLOAT BINARY items, while fractional digits are lost in FIXED DECIMAL computations. FIXED BINARY digits are lost in the most significant portions. A number of arithmetic conversion functions are available to control the conversions which take place during expression evaluation, as detailed in the following sections.

7.2.2. The FIXED Built-in Function. The FIXED Built-in function is referenced as FIXED (X) or FIXED (X,p) or 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 DECIMAL, the result is FIXED DECIMAL. Otherwise the result is FIXED BINARY. If X is FLOAT BINARY, the result is FIXED BINARY. If p or q is not specified, then the result is dependent 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).

7.2.3. The FLOAT Built-in Function. The FLOAT Built-in function is referenced as FLOAT (X) or FLOAT (X,p) where X is the variable or expression to be converted to a FLOAT arithmetic data type, and p gives 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),24)).

7.2.4. The BINARY Built-in Function. The BINARY Built-in function is referenced as BINARY(X) or 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, then the result is FLOAT BINARY. If p is not included 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)).

7.2.5. The DECIMAL Built-in Function. The DECIMAL Built-in function is referenced as DECIMAL(X) or DECIMAL(X,p) or 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 or the target result. If p and q are not included, 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).
```

7.2.6. The DIVIDE Built-in Function. The DIVIDE built-in function is used to control the precision of results for divide operations. The 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. The DIVIDE function is required for FIXED BINARY division since, in the full language, a non-zero scale factor results from such an operation.

7.3. String Conversions.

Conversions take place between arithmetic and bit string data items when they are combined in expressions. The various conversion rules for string operands are given in this section.

7.3.1. Arithmetic to Bit String Conversion. Conversion from an arithmetic source data type X to a bit string target takes place according to the following rules. The ABS(X) is converted to a FIXED BINARY (p) according to the arithmetic conversion rules. The FIXED BINARY intermediate value is converted to a bit string of length p. If the target length is longer than p, the intermediate result is padded on the right with zero bits. If the target length is less than p, the rightmost excess bits of the intermediate result are truncated.

7.3.2. Arithmetic to Character Conversion. The various arithmetic data types are converted to intermediate charararacter strings in the following manner. FIXED 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 zeroes, 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, conversion of a FIXED DECIMAL (3) with value 33Ø results in the character string 'bbb33Ø', where b denotes a blank position. The value zero produces a single zero digit result.

FIXED DECIMAL (p,q), q > Ø: the resulting character string is also of length p+3, is of the same string format stated above, except that the decimal point and the fractional digits are included. For example, conversion of a FIXED DECIMAL(5,2) data item with value -13.25 results in the character string 'bb-13.25'. Leading zeroes are omitted except for the one immediately preceding the decimal point.

FIXED BINARY (b): the source is converted to FIXED DECIMAL (p), where p = CEIL(b/3.32)+1. The FIXED DECIMAL (p) result is then converted to a character string of length p+3 with the format described above. Conversion of a FIXED BINARY(15) with value -32 results in 'bbbbbb-32'.

FLOAT BINARY (b): the fractional part is converted to a FIXED DECIMAL (p), where p=CEIL(b/3.32). The resulting character string is of length p+6 with the following standard scientific notation format. 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 digits 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 two digit exponent value. Conversion of a FLOAT BINARY (24) with value 25Ø.1E1 results in the character string 'bbbb2.5Ø1E+Ø3'.

If the target length is greater than the length of the intermediate result, the string is padded on the right with blanks. Conversely, if the target length is shorter than the intermediate result, the string is truncated on the right to produce the shorter length.

7.3.3. Bit String to Arithmetic Conversion. A bit string of length n, where Ø < n <= 15, when converted to an arithmetic data type is first converted to its FIXED BINARY (15) equivalent, then converted to the target value according to the rules discussed in the previous sections. '1Ø11'B converted to FIXED BINARY (15) yields the value 11.

7.3.4. Bit to Character String Conversion. A bit string of length n is converted to a character string of length n, where a zero bit is converted to a character zero, and a 1 bit is converted to a character one. If the target length is longer than the source, the target is padded on the right with blanks. If the target length is shorter than the source length, the excess rightmost characters are truncated.

7.3.5. Character to Arithmetic Conversion.

In the case of character to arithmetic conversion, the character string source must contain a valid arithmetic constant value. If X is a character string, conversions applied to X are affected by the arithmetic conversion built-in functions as follows.

FIXED (X) or DECIMAL (X) returns a FIXED DECIMAL value. If p is not given, then 15 is assumed. BINARY(X) produces a FIXED BINARY value. If p is not specified, it will be set to 15. Note that the result is only the integer porion of X. FLOAT(X) produces a FLOAT BINARY value. If p is not given, p = 24 is assumed. If X is null or contains all blanks, the converted value is zero.

The ERROR condition is raised 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:

Character	Target	Result
'ØØ987'	FIXED BINARY(15)	987
'9.87'	FIXED DECIMAL(6,2)	ØØØ9.87
'-9.87E2'	FLOAT BINARY(24)	-9.87E2
'-9.87E2'	FIXED DECIMAL(9,2)	ØØØØ987.ØØ
'-9.87E2'	FIXED DECIMAL(5,Ø)	ØØ987
'-987.372'	FIXED DECIMAL(4,2)	ERROR
'2I3'	FIXED BINARY(15)	ERROR

7.3.6. Character to Bit String Conversion.

In the case of character to bit conversion, the source character string must contain only the characters Ø and 1. Each Ø character is converted to a zero bit, and each 1 character is converted to a 1 bit. If the target length is greater than the source length, then padding occurs on the right with zeroes. If the target length is shorter than the source length, then truncation on the right occurs. If the source is the null string, or contains all blanks, then the result is a bit string of zeroes.

7.4. Pseudo-Variables.

Two built-in names, SUBSTR and UNSPEC, are predefined in all PL/I-8Ø programs, and can be used as source operands in expressions, or target operands on the left of assignment statements. These names are referred to as pseudo-variables since they appear to act like simple program variables.

7.4.1. Character SUBSTR. The character substring operator allows access to the individual characters within a string, and takes the two forms shown below.

```
        SUBSTR(char-variable,i)   and   SUBSTR(char-variable,i,j)
```

where char-variable is a CHAR or CHAR VARYING subscripted or unsubscripted variable reference, and i and j are FIXED BINARY expressions. When SUBSTR appears in an expression, the first form extracts the substring starting at position i for the remainder of the string, where the first character position is numbered as 1. The second form shown above performs the same function as the first, but the length of the extracted substring is j. Note that the result is undefined if either i or i+j exceeds the string length, where the length is the declared fixed size for CHAR variables, and the current length for CHAR VARYING variables.

When SUBSTR appears on the left of an assignment, it must appear alone. That is, the SUBSTR operation cannot be embedded in a string expression when it serves as the target of a string assignment. The SUBSTR operation 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 either the length of the char-exp, or the end of the char-variable, whichever occurs first. The second form has the same effect, except the field width which receives the characters is restricted to length j. Note that 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, although this may occur in various other implementations of PL/I. The following assignment shows the use of SUBSTR.

```
          SUBSTR(C(I),J,K+2) = SUBSTR(D,J) || SUBSTR(E,J+5,3);
```

7.4.2. Bit SUBSTR. Bit substring operations in PL/I-80 are similar to the character SUBSTR shown above, with some restrictions. First, PL/I-80 bit strings are limited to the precision range 1 through 16, corresponding to single and double byte values. In order 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)  and SUBSTR(bit-variable,i,k)
```

where the bit-variable is a subscripted or unsubscripted BIT variable reference, k is a literal constant in the range 1 to 16, and i is a FIXED BINARY expression. The effect of the SUBSTR operation is identical to the character operation described above, except a bit string of length k is selected when SUBSTR appears in an expression, and is assigned when SUBSTR appears on the left as a target of a bit string store operation. An example of bit SUBSTR is given in the following section.

7.4.3. UNSPEC. The UNSPEC pseudo-variable allows access to single and double byte variables as if they are 8 and 16 bit bit string data items. The form of an UNSPEC reference is:

```
UNSPEC(variable)
```

where the variable is a subscripted or unsubscripted variable reference to a data item which occupies a single or double byte memory location. Note that a temporary result is not allowed as an argument. When UNSPEC is used in an expression, the result is the bit string value of the argument. When UNSPEC appears on the left of an assignment, the assigned value is converted to bit string and directly stored into the single or double byte value.

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. Be aware, however, that novic programmers often fall into the trap of using UNSPEC instead of a more appropriate high level language facility. In fact, whenever it seems necessary to use UNSPEC, one should examine the problem in a more general way to see if its use could be avoided. The following example shows a case where two memory locations are being accessed. The UNSPEC operation is used to load 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.

```
DCL
    (P, Q) POINTER,
    A BIT(16) BASED (P),
    B BIT(8) BASED  (Q),
    I FIXED;
    . . .
    UNSPEC(P) = 'FF80'b4;
    UNSPEC(Q) = 'FFF0'b4;
    . . .
    SUBSTR(B,4,2) = SUBSTR(A,I,2);
    . . .
```

8. SEQUENCE CONTROL STATEMENTS

Sequence control refers to the order in which the statements of a program are executed. Program statements are normally executed sequentially with interspersed sequence control statements which alter this normal flow. Generally, sequence control statements allow conditional and unconditional branching and controlled looping, as discussed below. Procedure invocations, which also alter the normal sequence execution, are covered later in this section.

8.1. The GO TO Statement.

The GO TO statement causes unconditional transfer to a specific labelled statement, and takes one of the forms shown below:

```
GO TO label-constant;
GOTO  label-constant;
GO TO label-variable;
GOTO  label-variable;
```

where the label-constant is a literal label which appears as the prefix of some labelled statement, and label-variable is a simple or subscripted label variable which has been assigned the value of a label-constant. The evaluated label-constant must label a statement within the scope of the GO TO statement, and cannot be within an embedded DO group of any sort. Three examples of GO TO statements are given below.

```
GO TO LAB1;
GOTO WHERE;
GO TO L(J);
```

8.2. 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 statement group to execute when the boolean test produces a false value. The general form of an IF statement is shown below.

```
IF condition THEN group-1; [ELSE group-2];
```

where the condition is a scalar expression yielding a bit string value, while group-1 and group-2 are either simple statements, or compound statements contained within a DO or BEGIN group. If either group-1 or group-2 is a simple statement, it cannot be a DECLARE, END, ENTRY, FORMAT, or PROCEDURE statement. If any bit of the result of the condition is 1 the THEN group-1 is executed, otherwise control passes to group-2, if included, or to the next statement in sequence following the IF statement.

IF statements may themselves be nested. In this case, each ELSE is paired with the innermost unmatched IF-THEN pair. Empty statements can be used to force the desired IF-ELSE pairing. Consider the following nested IF statements

```
IF A = Y THEN
    IF Z = X THEN
        IF W > B THEN
            C = Ø;
        ELSE C = 1;
    ELSE;
ELSE A = Y2;
```

where the null statement following the third ELSE corresponds to the second IF-THEN test.

8.3. The Iterative DO Statement.

In its simplest form, the DO group is just a collection of statements executed once. This non-iterative form of the DO group was presented in an earlier chapter. This section describes the iterative DO group, which is headed by a DO statement of two general forms:

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

where the control-variable is an unsubscripted variable, the condition is a boolean expression, and the do-specification is one of:

```
[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 which replaces the control variable after each iteration, and condition is an expression yielding a bit string value which is considered true if any of the bits in the string are 1. If the TO end-exp form is included, but the BY incr-exp is omitted then the incr-exp is assumed 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.

The WHILE expression is evaluated before execution of the

DO-group. The loop execution terminates, and control passes to the statement following the balanced END if the condition is false.

With the exception of the REPEAT option, expressions in the DO specification are evaluated before execution of the loop, so that changes made to the start, end, or incremental values do not affect the number of times a loop is executed. In the case of the REPEAT option, however, the repeat-exp is recomputed after each iteration. This recomputed expression is stored into the control-variable and the WHILE test, if included, is evaluated.

In order to properly define the actions of iterative groups, they are decomposed below into a sequence of equivalent IF and GO TO statements. In the decomposition, 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. To begin with, the DO-WHILE statement

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

can be expressed as the equivalent sequence of statements:

```
     LOOP:
          IF ^e1 THEN
              GO TO ENDLOOP;
          . . .
          GO TO LOOP;
     ENDLOOP:;
```

Similarly, the DO-REPEAT group

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

becomes

```
          i = e1;
     LOOP:
          . . .
          i = e2;
          GO TO LOOP;
```

Note that in this case, the loop proceeds indefinitely, until terminated by an embedded statement, such as a GO TO 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
                         GO TO ENDLOOP;
                     . . .
                     i = e2;
                     GO TO LOOP;
                ENDLOOP:;
```

The simple iterative DO-group

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

is treated as

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

which can be expressed as the equivalent sequence:

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

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

```
                     IF i < LAST THEN
                         GO TO ENDLOOP;
```

Otherwise, the test becomes

```
                     IF i > LAST THEN
                         GO TO 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
                 GO TO ENDLOOP;
              IF endtest THEN
                 GO TO ENDLOOP;
              . . .
              i = i + INCR;
              GO TO LOOP;
         ENDLOOP:;
```

Note that in these equivalent sequences, the value of LAST and INCR take on the characteristics of the expressions e2 and e3. Further, arithmetic conversions and comparisons take place at each step according to the normal PL/I-80 rules.

8.4. Condition Processing.

The ON, REVERT and SIGNAL statements provide run-time facilities for programmatic interception of error conditions which, in most cases, would cause termination of the program. The following conditions are recognized by PL/I-80: a general error condition (ERROR), certain computational error conditions (FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVIDE), and certain I/O conditions (ENDFILE, ENDPAGE, KEY, and UNDEFINEDFILE). Additional details concerning exception processing are given below, and in the "PL/I-80 Applications Guide."

8.5. The ON Statement.

The ON statement defines the condition to handle and the action to take when the condition is raised during program execution. The ON statement has the format:

```
ON condition ON-unit
```

where the condition may be any one of the following:

```
      ERROR   FIXEDOVERFLOW   OVERFLOW   UNDERFLOW
ZERODIVIDE   ENDFILE   ENDPAGE   UNDEFINEDFILE   KEY
```

The ON-unit may be a PL/I-80 statement, or several PL/I-80 statements contained within a BEGIN-END block which are executed when the particular condition named in the ON statement is raised. Exit from the BEGIN block may not be through a RETURN statement, although this restriction does not preclude procedure definition within the BEGIN-END block. Transfer out of the BEGIN block takes place through a non-local GOTO statement. Control resumes at the point where the condition was signalled if all the statements of the BEGIN-END group are executed and no non-local transfer has occurred.

An ON-unit may not free a variable that is being used when the condition is raised, or close the file for which the I/O condition is raised. The ON-unit remains active until its encompassing block is terminated or until it is reverted by the REVERT statement or another ON-unit is established later in the execution sequence. In this latter case, the ON-unit is stacked until reactivated through a REVERT statement which cancels the most recent ON-unit. Note that a maximum of sixteen ON conditions may be active at any given point.

8.6. The SIGNAL Statement.

The SIGNAL statement causes a particular condition to be programmatically raised, and invokes the corresponding ON-unit, if active. If no ON-unit is active, the default system action takes place. In most cases, the default action prints a "backtrace" and terminates program execution. The form of the SIGNAL statement is:

```
SIGNAL condition;
```

where the condition is one of those listed above with the ON statement. For example,

```
SIGNAL ZERODIVIDE;
```

invokes the current ZERODIVIDE ON-unit.

8.7. The REVERT Statement.

The REVERT statement is used to deactivate the current ON-unit and re-establish the one which preceded it, if it exists. The format of the REVERT statement is:

```
REVERT condition;
```

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

```
REVERT OVERFLOW;
```

deactivates the current ON-unit for the OVERFLOW conditon. Note that upon exit of a PROCEDURE or BEGIN block, an automatic REVERT statement takes place for any ON-units enabled within the block.

8.8. Default ON-Units.

With the exception of FIXEDOVERFLOW and ENDPAGE, the default ON-units normally print an appropriate error message followed by program termination. The FIXEDOVERFLOW condition is not signalled for FIXED BINARY overflow, although it will occur if FIXED DECIMAL computations exceed their allocated field sizes. When the ENDPAGE condition is raised, the default ON-unit inserts a form feed character into the output file and sets the current line number to 1.

8.9. Built-In Functions for Condition Processing.

PL/I-8Ø includes several built-in functions designed to aid in exception handling, as listed below.

```
        ONCODE   ONFILE
     ONKEY   PAGENO   LINENO
```

The ONCODE function returns a FIXED BINARY value representing the type of error which raised the most recent ERROR condition. If no condition was raised, then the returned value is zero. The error codes vary from version to version, and may be found in the "PL/I-8Ø Command Summary."

The ONFILE, ONKEY, PAGENO, and LINENO built-in functions are presented in detail in a later chapter.

8.1Ø. Procedure Blocks.

Procedure blocks, delimited by balanced PROCEDURE and END statements, are invoked by subroutine CALL statements or by function calls. Procedures are used to execute the same program segment one or more times without duplicating the segment several times throughout the program. Communication data transmitted to the procedure are referred to as actual parameters, while the list of variable expected by a procedure, and defined in the PROCEDURE statement, are referred to as formal formal parameters.

8.11. Invoking a Procedure.

There are two types of procedures: subroutine procedures and function procedures. The difference between a subroutine and a function is that the subroutine is invoked through a CALL statement while the function is invoked by simply using its function name followed by a set of parentheses enclosing the actual parameters, if any, where the context requires an expression. Further, a function procedure returns a scalar value to the calling program segment.

The CALL statement is used to transfer control to a subroutine procedure and pass information (if necessary) to the procedure. The format of the CALL statement is:

```
CALL procname [(subl,...,sub-n)] [(argl,...,arg-m)];
```

where the procname is the name of the procedure being invoked and subl through sub-n represent a list of optional subscripts which are required if procname is a subscripted entry variable. The elements argl through arg-m represent the actual parameters passed to the procedure. Actual parameters may be any of the following: an arithmetic or string variable, an array or structure, a constant, an expression, a label, a file name, or a pointer. Cross-sections of a multi-dimensional array cannot be used as actual parameters. Note that the number of actual parameters is defined by the corresponding PROCEDURE heading and, if no parameters are required, an empty pair of parentheses must be given. Examples of CALL statements are:

```
          CALL P();
       CALL Q(A,(B),B+C);
    CALL V(I,J) (A,(B),(B+C));
```

A function is invoked in the same manner, except that the CALL keyword is not required, and the resulting value must be used as an expression. Examples of function invocations are shown below:

```
          I = F();
     IF G(A,(B),B+C) = 5 THEN
    I = H(I,J)(A,(B)) + SQRT(X);
```

8.12. The Structure of a Procedure Definition.

The previous section described the method by which a procedure is invoked. The purpose of this section is to describe the overall structure of a subroutine or function definition. Procedures may be defined at any point in a particular program, and are bypassed when encountered in the sequential control flow. That is, procedures are only activated through the invocation mechanisms described in the previous section. Normally, all procedures are defined together in a single section at the beginning or end of the main program, depending upon programming style. The main program is, itself, a single procedure definition. Further, separate procedures can be defined and linked together to form a single module. The overall structure of any procedure definition is

```
procname:
     procedure-statement
     procedure-body
     END [procname];
```

where procname is an identifier which names the procedure. The procedure name becomes an entry constant which is available for access throughout the scope in which it appears. The entry point to a procedure is identified by the procedure-statement. The procedure-body may consist of a sequence of zero or more statements. The procedure definition is terminated by a corresponding END statement which may also be the exit point of the procedure, although embedded RETURN statements may appear within the procedure-body.

The procedure-statement delimits the beginning of the procedure block, defines the formal parameter list, and gives the attributes of the returned value for functions. The general format is

```
          PROCEDURE [(parml, ..., parm-n)]
[OPTIONS(MAIN)] [RETURNS attribute-list] [RECURSIVE];
```

where parml through parm-n are the formal parameters for the procedure. All formal parameters must be declared within the procedure body at the principal block level. A formal parameter may be one of the following: a non-subscripted variable an array, a major structure, a label variable, a file name, a entry name, or a pointer variable. A formal parameter may not have the following attributes:

```
STATIC    AUTOMATIC    BASED    EXTERNAL
```

OPTIONS(MAIN) identifies this procedure as the first procedure to receive control when the program starts. The RETURNS attribute-list is required for a function procedure to give the characteristics of the value returned by the function. The RECURSIVE attribute indicates that this procedure may activate itself, either directly or indirectly, while the procedure is being executed.

8.13. The RETURN Statement.

The RETURN statement returns control to the point in the calling block right after the procedure invocation and returns a value as well if the procedure is a function procedure. The format of a RETURN statement is:

```
RETURN [(ret-exp)];
```

where ret-exp is the value which is returned to the calling point. If necessary, the returned value is converted to conform to the attributes specified in the RETURNS option of the procedure-statement. Examples of RETURN statements follow:

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

The RETURN statement terminates the procedure block which contains it. If the RETURN statement is in the main procedure, control returns to the operating system.

8.14. The Non-Local GO TO Statement.

A non-local GO TO statement occurs when the evaluated target label constant occurs outside the innermost block containing the GO TO statement. In general, the non-local GO TO should be avoided since poorly-structured programs often result. There are occasions, however, when the non-local GO TO is appropriate. In particular, in the case of terminal error conditions, it is often useful to branch directly to a global error recovery label where program execution recommences. In this case, all embedded ON-units are automatically reverted, and procedure return information is discarded. The following skeletal program shows an instance of a non-local GO TO from within a procedure definition:

```
P:
   PROCEDURE;
   GO TO L;
   END P;
CALL P();
L:;
```

8.15. The STOP Statement.

The STOP statement terminates execution of the program, closes all open files, and returns control to the operating system. Normally, the STOP statement occurs only at the main program level, but may be executed within a nested procedure call to prematurely halt execution. The format is simply:

```
STOP;
```

8.16. Arguments and Parameters.

As stated previously, data items passed by the invoking block are referred to as actual parameters and data items expected by the invoked procedure are referred to as formal parameters. Upon invocation, each actual parameter is paired with its corresponding formal parameter. When the actual parameter and corresponding formal parameter share storage, the actual parameter is said to be passed by reference. In this case, any changes made to the formal parameter in the invoked procedure changes the value of actual parameter of the invoking block.

When the actual and formal parameters do not share storage, the actual parameter is said to be passed by value. In this case, a copy of the actual parameter is sent to the invoked procedure, so that any changes to the formal parameter affect only the copy, not the actual parameter value.

Arguments passed by reference are those variables whose attributes conform to those of the formal parameters. Aggregate expressions, consisting of arrays and structures, must always conform in subscript range, type, precision, and scale, and thus are always passed by reference.

An actual parameter is passed 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, or an expression which does not conform to the formal parameter specification. In the latter case, the actual parameter is converted to the type, precision, and scale of the formal parameter. Given that the procedure P begins with the statements

```
P:
    PROCEDURE(A,B,C);
    DCL
        A CHAR (1Ø),
        B FIXED,
        C FLOAT;
    ...
```

a CALL of the form shown below sends three actual parameters to the procedure, corresponding to the three formal parameters A, B, and C. In this example, assume X is CHAR(1Ø), and Y and Z are both FIXED:

```
CALL P(X,(Y),Z);
```

The first actual parameter X is passed by reference since it matches the formal parameter A. The second parameter is passed by value since it occurs as an expression. The third parameter is converted to FLOAT type and passed by value.

8.17. The ENTRY Attribute.

Entry data items are used to identify procedure names, and consist of entry constants or entry variables. Entry constants correspond to internal procedures, or separately compiled external procedures. Entry variables are data items which may take on entry constant values during program execution.

The characteristics of the formal parameters and returned values for externally compiled procedures must be defined in the calling program with an ENTRY declaration. It is essential that the programmer ensure that the entry declaration properly matches the externally defined procedure, so that the proper linkage takes place when the program segments are combined with the link editor. Further, variables which take on entry constant values are also defined with an ENTRY declaration. Entry variables may be subscripted, if required by the application. The ENTRY attribute is used to define an identifier as an entry data item, and gives the attributes of the formal parameters, as well as the optional returned value attributes if the entry item is a function. The format for an ENTRY declaration is:

```
                 DECLARE procname [(sub1,...,sub-n)]
                [VARIABLE] [ENTRY [(att1,...,att-m)]]
                      [RETURNS (ret-att)];
```

where the attributes may be listed in any order, and at least one of ENTRY or RETURNS must be listed. The identifier given by procname is the entry data item name, sub1,...,sub-n is the optional subscript list, att1,...,att-m is the list of formal parameter attributes, and ret-att is the optional returned value attribute for a function entry item. The VARIABLE attribute indicates that the data item is an entry variable which must be assigned an entry constant value during program execution. Note that the subscript list is only valid if item has the VARIABLE attribute, and the list of formal parameter attributes is omitted if no parameters are required by the procedure. In this case, the ENTRY attribute can also be omitted if the RETURNS attribute is present.

If the dimension attribute is specified for a particular parameter, it must be the first attribute declared. If the formal parameter is a structure, the structuring information is specified by level numbers preceding the attribute definition. Attribute factoring is not permitted within the list att1 through att-m. Examples are given below:

```
                         DECLARE X ENTRY;
                     DECLARE Y ENTRY VARIABLE;
             DECLARE P (Ø:1Ø) ENTRY (FIXED,FLOAT) VARIABLE;
          DECLARE Q ENTRY (1, 2 FIXED, 2 FLOAT, (5:1Ø) DECIMAL);
                    DECLARE R RETURNS (CHAR(1Ø));
```

9. INPUT/OUTPUT PROCESSING

This chapter presents the input/output facilities of PL/I-8Ø which allow data transmission between memory and external devices. A considerable portion of the overall power of PL/I is found in its I/O processing facilities. Thus, a more complete treatment of the application of I/O features is found in the accompanying document "PL/I-8Ø Applications Guide." For this reason, the discussion below is intended to be spartan and definitive, rather than explanatory. Please reference the accompanying manuals for complete examples.

9.1. FILE Data Items.

An external device may be a console, a line printer, or a disk file. The collection of data elements transmitted to or from an external device is referred to as a data set, while a corresponding internal file constant or variable is referred to as simply a file. Except for the predefined standard input and output files, called SYSIN and SYSPRINT, all files accessed in a particular program must appear in a file declaration of the form

```
DECLARE fname FILE [VARIABLE];
```

where fname is the file identifier. The declaration defines a file constant if the VARIABLE option is not included. If the VARIABLE attribute is given, then the declaration defines a file variable which can take on the value of a file constant through an assignment statement.

A file constant declaration creates a file parameter block which is a segment of memory containing information about the file, as described in a following section. A file variable declaration does not cause the creation of a parameter control block. Further, a file variable is valid in input/output operations only after it has been assigned a file constant. The equal and not equal comparison operators may be used with file data, the items are equal if they refer to the same file parameter block. Finally, note that a file constant is given the EXTERNAL attribute, while a file variable is local to the block in which it is declared unless it is declared EXTERNAL.

9.2. File Types.

PL/I-8Ø recognizes three basic file types: stream-oriented, record sequential, and record direct files. The file type determines the how data is stored and how it is transmitted or accessed.

In stream I/O data is treated as a sequence of ASCII characters organized into lines and pages. Lines are separated by a linemark and pages are separated by a pagemark. A linemark is a carriage return, line feed pair, or just a single line feed, while a pagemark is a form feed character. A stream file is accessed only sequentially. That is, data items are read or written in order until the end of the file is reached or until the file is closed.

Both formatted I/O and free-format I/O are available for stream files. When stream data is being transmitted to memory, the input characters are converted to the data type of the receiving variable. Conversely, data being transmitted to a stream file is converted to its ASCII representation. Data type conversion rules apply to I/O conversion, as described in an earlier section.

The size of the data item transmitted during record I/O varies depending upon the item size and the file characteristics. No conversion, however, occurs during record transmission resulting in a transfer of the binary bit patterns which represent the data items in the memory of the machine.

Record sequential files may be accessed only sequentially. That is, records are read or written in linear order. Record direct files, on the other hand, are accessed according to key values supplied in the READ or WRITE statement, and need not be sequential. Each record in a record direct file has an associated key value which provides a unique identification of the record for subsequent access.

9.3. Opening a File.

A file must be opened before any I/O transactions take place on the data set. A file may be opened explicitly, through the use of the OPEN statement, or implicitly when the file is accessed through a GET, PUT, READ, or WRITE statement. When the file is opened the following occurs. First, the file's attributes are consolidated. Next, the file is associated with an external data set. If the file is an input file and no external data set exists, the UNDEFINEDFILE condition is raised. If the file is an output file, any existing data set by the same name is erased, and a new data set is created. Similarly, if the file is opened for update access, the data set is created if it does not exist. No action takes place if the file is already open.

The OPEN statement is used to explicitly open a file. The format of the OPEN statement is:

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

where fname is the file name which appears in a FILE declaration statement, and file-attributes denotes any valid combination of the following keywords:

```
        STREAM RECORD INPUT OUTPUT KEYED DIRECT
    SEQUENTIAL UPDATE PRINT TITLE PAGESIZE LINESIZE
```

The interpretation of the OPEN statement and the use of the keywords shown above are somewhat implementation dependent. Thus, the complete details of each keyword format, along with associated parameter values, is given in detail in the accompanying Digital Research manual entitled "PL/I-8Ø Applications Guide."

9.4. The File Parameter Block.

Each file constant provides access to a collection of values stored in a File Parameter Block (FPB). Each FPB contains the information listed below. The file status indicates whether the file is open or closed. The file title names the peripheral or data set associated with the file constant. The column position is maintained, in order to locate the next position to get or put data in a STREAM file. The current line is counted in STREAM files, as well as the current page for PRINT files. The current record position is also maintained. Upon opening a file, the FPB additionally addresses the CP/M File Control Block (FCB), as well as the line size, page size, fixed record size, internal buffer size, and file descriptor. Additional implementation-dependent information is also accessible from the FPB. Following a successful OPEN operation, the file descriptor defines one of the following sets of attributes for the file:

```
         STREAM  INPUT
         STREAM OUTPUT
    STREAM   OUTPUT   PRINT
    RECORD INPUT  SEQUENTIAL
    RECORD OUTPUT SEQUENTIAL
 RECORD INPUT  SEQUENTIAL KEYED
 RECORD OUTPUT SEQUENTIAL KEYED
 RECORD   INPUT   DIRECT  KEYED
 RECORD  OUTPUT   DIRECT  KEYED
 RECORD  UPDATE   DIRECT  KEYED
```

9.5. Input/Output ON Conditions.

A number of conditions may be raised during I/O processing. The ENDFILE condition is raised whenever an input operation reads past the end of file, which is a control-z for STREAM files and a physical end of file for RECORD files. The ENDFILE condition is also raised for OUTPUT files when all disk space is exhausted.

The ENDPAGE condition is raised for a STREAM output file with the PRINT attribute when the line number exceeds the page size. The default ON-unit sends a form feed to the output, and resets the line number to one.

The KEY condition is raised whenever an invalid KEY is detected. This will occur in PL/I-8Ø if the FIXED BINARY key value, times the fixed record size exceeds the capacity of the disk.

The UNDEFINEDFILE condition is raised whenever an input file does not exist, or an output or update file cannot be created due to insufficient directory space. This condition is also raised if the file name provided in the TITLE option is improperly formulated.

A number of built-in functions are available which aid in I/O exception processing, and are described in detail in a later section:

LINENO   PAGENO   ONFILE   and   ONKEY

Again, please refer to the "PL/I-8Ø Applications Guide" for additional implementation-dependent information.

9.6. The CLOSE Statement.

The CLOSE statement disassociates the file from the external data set. The format of the CLOSE statement is:

```
CLOSE FILE(fname);
```

where the evaluation of fname produces a file constant. operation is applied. If the file does not have an open status, the close operation is ignored. Otherwise, the buffers are cleared and output files are permanently recorded on the disk. The file may subsequently be reopened using the OPEN statement described above.

9.7. Predefined Files.

The file constants SYSIN and SYSPRINT are a part of all PL/I-8Ø programs, and need not be declared unless an explicit file reference takes place in an OPEN, GET, PUT, READ, or WRITE statement. In the case that the FILE option is not given in a GET, or READ statement, the file SYSIN is automatically accessed. Similarly, the file SYSPRINT is accessed in PUT or WRITE statements without the FILE option. In these cases, the SYSIN file becomes the console keyboard, with a linesize of 8Ø characters, while the SYSPRINT file becomes the console output display with line size 8Ø, and infinite page size.

10. STREAM ORIENTED INPUT/OUTPUT

STREAM files are made up of a sequence of ASCII characters separated by linemarks and pagemarks. STREAM I/O statements provide the facilities for accessing character data in a STREAM file. In general, the following rules apply to 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, otherwise, if the input or output character is graphic, the column position is advanced by 1. If, on output, the column position exceeds the linesize, a linemark is written, the line number is incremented by 1, and the column position is reset to 1. When the line number exceeds the page size, a pagemark is written, and the column position and line number are reset to 1.

Three forms of STREAM I/O are provided in PL/I-80, called list-directed, edit-directed, and line-directed. List-directed I/O transfers data items without format specifications. Edit-directed I/O allows formatted access to character data, while line-directed I/O allows access to variable length character data in an unedited form. Note that line-directed I/O is provided in PL/I-80 to process variable length ASCII records using READ and WRITE statements and may not be available in other versions of PL/I.

The following naming conventions are used in the descriptions of the various STREAM I/O statements:

fname	is the file identifier.
nl	is a FIXED BINARY expression which defines the number of linemarks to skip on input, or the number of linemarks to write preceding the data item on output.
input-list	is a list of variables separated by commas, to which the data items from the input stream are transmitted. The input-list determines the number and order of the variables assigned by the input data in the stream. The variables must be scalar values in PL/I-80. iterative DO loops may be included in the input-list. The DO header format follows that of of the DO statement described previously, with the exception of the REPEAT clause which is not included. The general format is (item-1,...,item-n DO iteration) as in the statement GET LIST ((A(I),B(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 may also include iterative DO groupings, as shown above in the input-list.

10.1. LIST-Directed I/O.

The input stream for list-directed I/O must have the following properties. Data items in the stream may be arithmetic constants, character string constants, or bit string constants. Each data item must be followed by a separator, consisting of a series of blanks, a comma surrounded optionally by blanks, or an end of line. Embedded tabs (ctl-I) are treated as blanks. Character string data which actually contain blanks or commas must be enclosed in quotes. Otherwise, the blanks or commas will be treated as separators.

A null field in the input stream is indicated by a comma as the first non-blank character in the input line, or by two consecutive commas optionally separated by one or more blanks. The null field indicates that no data is to be transmitted to the associated data item in an input-list, and thus the value of the target data item remains unchanged. Linemarks are counted when the SKIP option appears in GET statement, but otherwise serve only as separators.

10.2. The GET LIST Statement.

The GET LIST statement is used to read data using list-directed I/O. The format of the GET statement is:

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

At least one of the options must be given, and may appear in any order, except for the LIST option which must appear last. If the FILE option is omitted, FILE(SYSIN) is assumed. If nl is not listed with the SKIP option, then one linemark is skipped.

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

Character strings in the input stream may or may not be enclosed in quotes, but if included, the enclosing quotes are not transmitted to the input variable. Likewise, for bit string constants, the enclosing quotes and the trailing B are not transmitted to the input variable.

Input strings are limited to one line. Thus only the leading quote is necessary for string input from the console when terminated by a carriage return.

10.3. The PUT LIST Statement.

The PUT LIST statement is used to write data using list-directed I/O. The format of the PUT LIST statement is:

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

As in GET LIST, at least one of the options must appear, and the LIST option must appear last. If the FILE option is omitted, FILE(SYSPRINT) is assumed. If the SKIP option is used and nl is not specified, then nl defaults to 1. If nl = Ø no linemark is written but the column position is reset to 1. In any event, any time the SKIP option is used the column position is reset to 1. The PAGE option is valid only for PRINT files. If p is not specified it defaults to 1. Note that whenever a pagemark is written, both the column position and line number are set to 1. The data items in the output-list are converted to their character string representation and written to the STREAM file. Blanks are used to separate the data on the output file. If, however, the data item is longer than the number of characters left on the output line, the item will be written at the beginning of the next line. If the length of the character string representation of the data item exceeds the line size, the data item is written by itself on a single line which extends past the line size. If the page size is exceeded during output transmision, the ENDPAGE condition is raised.

Normally, character strings are written within enclosing quotes, with each embedded quote symbol written as a pair of quotes. If the file has the PRINT attribute, the additional quotes are omitted. Bit string data is always written within enclosing string quotes, followed by the letter B.

10.4. EDIT-Directed I/O.

The input-list and the output-list for edit-directed I/O are the same as those for list-directed I/O. However, the manner in which the data are read or written is determined by a list of format items in the format-list of the GET EDIT and PUT EDIT statements.

10.5. The FORMAT-List. The format-list is a list of format items, separated by commas, that describe the data items to be read (data format items), specify the placement of the data items in the stream (control format items), or reference another format-list (remote format item). The general form of a format-list is:

```
[n] f-item ...[,[n] f-item]
```

where n is an literal constant value in the range 1 to 254 which gives the repetition factor of the following f-item. A repetition factor of one is assumed if n is omitted. The f-item is either a data format item or a control format item. In order to allow repetition of a number of format items, an f-item can also be a group:

(format-list)

An f-item can also be a remote format item. In PL/I-8Ø, however, a remote format item must be the only format in the list, and cannot be preceded by a repetition factor.

1Ø.6. Data Format Items.

Data format items are used to read or write numeric or character fields from or to an external STREAM data set. PL/I-8Ø supports the following data format items:

F(w,[d])

Used for fixed point arithmetic data, where 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.

On input, as many characters as specified by w is read. If the character string contains a decimal point that decimal point determines the scale. Otherwise, d determines the scale. Leading and trailing blanks are ignored. If the field contains only blank characters, the value read is zero.

On output, d specifies the scale of the output value. If d is omitted, the scale is zero. The output value is rounded unless the variable has precision 15 (the maximum precision). Leading zeroes are suppressed except for the one immediately preceding the decimal point.

E(w[,d])

Used on output to represent arithmetic data in scientific notation format. It is used on input to convert decimal characters to float binary values. w defines the field width, while d gives the number of digits to the right of the decimal point. On output, w must be at 7 more than d, since the output field will appear as

+n.ddddE+ee

where + represents sign positions, n is the leading digit, dddd represents the fractional part of length d, and E+ee represents the exponent field.

A[(w)] reads or writes w characters of character string data. On input, w must be included to be compatible with full PL/I. PL/I-8Ø, however, allows w to be omitted on input, in which case the remainder of the current line is read up to, but not including the carriage-return line-feed. On output, if w is omitted, then w is assumed to be the length of the output string. If w is greater than the output string length, then blanks are added to the right. If w is less than the output string length, the string is truncated in the rightmost positions.

B[b][(w)] indicates bit string data representation. On input w must be included, and only Ø's and 1's must be in the input stream, otherwise the ERROR condition is raised. The number of bits to be used for each digit is given by b. On output, the variable is converted to a bit-string type then converted to its character string representation. If w is not included, the resulting character string is output. If w is specified and is longer than the character string, then padding with blanks occurs on the right. If the resulting character string is longer than w the ERROR condition is raised.

1Ø.7. Control Format Items.

Control format items are used for line, page, and space placement. PL/I-8Ø supports the following control format items:

COLUMN(nc) or COL(nc) moves the format pointer to column nc in the input or output data stream.

On input, characters passed over by positioning to column nc are ignored. If the current column position is less than nc the format pointer moves column position nc. If the current column position is greater than nc, the pointer moves to the next line, 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. On input, movement of the format pointer discards input characters. On output, blanks are added to the stream as the pointer is advanced.

On output, blanks are written in the process of positioning to column nc. Also, if the current position is greater than nc, the program outputs a linemark, then outputs blanks until it reaches column nc of the new line. If nc exceeds linesize, a linemark is written and the column position is set to 1.

X(sp)

Advances the format pointer sp positions in the input or output data stream.

On input, sp is the number of characters to be advanced. Linemarks are ignored when encountered, with the operation continued on the next line.

On output sp is the number of blanks to be written. If the end of the line is reached, a linemark is and the blank fill operation is continued on the next line.

SKIP[(nl)]

Specifies the number of linemarks nl to be skipped or written. If nl is omitted, 1 is assumed. The column position is set to 1.

On input, nl is the number of linemarks to skip before moving to the next format item. Note that if SKIP(1) is executed as the first format item immediately following an explicit or implicit OPEN operation, the first line is discarded. Further, SKIP(Ø) is undefined for input streams.

On output, nl is the number of linemarks to be written. If, in the process of writing linemarks, the pagesize is exceeded for a PRINT file, the ENDPAGE condition is raised and, upon return from the ON-unit, the SKIP operation is aborted.

LINE(ln)

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 has no effect. If the current line number is less than ln, then linemarks are output until the current line number equals ln. The ENDPAGE condition is raised if the sufficient linemarks are issued to exceed the current page size.

PAGE

Used only with PRINT files, the PAGE option causes a pagemark to be written, the page number incremented by one, the line number and the column position set to 1.

Note that control format items are executed as they are encountered in the format-list. Any control format items occurring after the input-list or output-list is exhausted have no effect.

10.8. Remote Format Items.

The remote format item uses the format-list of a FORMAT statment in place of the remote 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, which is in the scope of the remote format item. As mentioned above, PL/I-80 has the restriction that only the remote format item can appear only by itself in the format-list, with no preceding repetition factor, as shown below:

```
PUT SKIP EDIT(A,B,C) (R(ELSEWHERE));
```

10.9. The FORMAT Statement.

The FORMAT statement defines a remote format item, and appears as follows:

```
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 as 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 in the statement

```
GET EDIT (A,B,C) (R(L1));
```

10.10. The GET EDIT Statement.

The GET EDIT statement reads data using a format-list. The general form is:

```
GET [FILE(fname)] SKIP[(nl)] [EDIT(input-list)(format-list)];
```

Like the GET LIST, data items are read 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, however, pairs each input-list item with the next sequential format-list item, appying control-format items as they are encountered in the process. If the input-list is exhausted before the end of the format-list, remaining format items are ignored. If the format-list is exhausted before the end of the input-list, the format-list is reprocessed from the beginning.

10.11. The PUT EDIT Statement.

The PUT EDIT statement writes output data items according to a format-list. The form of the PUT EDIT statement is:

```
PUT [FILE (fname)] [SKIP [(n)]] [PAGE [(p)]]
      [EDIT(output-list)(format-list)];
```

Similar to the GET EDIT statement, the PUT EDIT pairs output expressions from the output-list with format items from the format-list. Control format items encountered during this process are applied. Unprocessed format items are ignored at the end of the PUT statement. Further, the format-list is restarted from the beginning if the end of the list is encountered during processing.

10.12. Line-Directed I/O.

Two forms of the READ and WRITE statement are available in PL/I-8Ø for processing variable length ASCII records in a STREAM file. These forms are not generally available in other implementations of PL/I, and thus should be avoided if upward compatibility is important. The two forms, called READ Varying and WRITE Varying, are described below.

10.13. The READ Varying Statement.

The READ statement may be used to read variable length STREAM INPUT files. The form of the READ Varying statement is

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

where v is a CHAR VARYING string variable. FILE(SYSIN) is assumed if the FILE option is not present.

It is important to note that the READ statement, discussed in a following section, is differentiated from the READ Varying form only by the fact that the target variable has the VARYING attribute.

Data is read from the file until the maximum of the length of v is reached, or a line-feed character is read. The length of v is set to the number of characters read, including the line-feed character. Note that when the READ Varying statement causes a default OPEN to occur, the resulting file attributes include STREAM INPUT. attributes STREAM INPUT. Given the declaration

```
1 BUFFER,
  2 BUFFCH CHAR(254) VAR;
```

for example, the statement

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

produces a RECORD oriented data transmission since the target is a structure, not a CHAR VARYING variable. The statement

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

however, is interpreted as an ASCII STREAM INPUT data transmission since the target is CHAR VARYING.

10.14. The WRITE Varying Statement.

The WRITE Varying statement is used to write variable-length ASCII STREAM data. The form of the WRITE Varying statement is:

```
WRITE [FILE(fname)] FROM (v);
```

where v is a CHAR VARYING string variable. No additional control characters are added to the output string. If control characters are requrired in the application, they must be included as a part of the string. Recall that PL/I-80 allows embedded control characters as a part of string constants, denoted by a preceding "^" within the string.

Again, WRITE Varying is not generally available in implementations of PL/I, and must be avoided if upward compatibility is a requirement. As in the READ Varying case, WRITE and WRITE Varying are differentiated by the fact that the source variable has the VARYING attribute. Further, note that the default OPEN produced by this statement includes the STREAM OUTPUT attributes. Given the declaration in the previous section, the statement

```
WRITE FILE(F) FROM(BUFFER);
```

is taken as a RECORD oriented data transmission, while

```
WRITE FILE(F) FROM(BUFFCH);
```

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

11. RECORD ORIENTED INPUT/OUTPUT

Record files contain binary data which is transmitted without conversion to or from connected storage. Two forms of RECORD processing are allowed: SEQUENTIAL, where records are accessed in the order they appear, and DIRECT, where records are accessed through keys. The various general forms are discussed in the sections which follow. In each case, fname is a file variable or file constant, x is a connected aggregate or scalar data type which is not CHAR VARYING, and k is a FIXED BINARY key value or variable. Again, most of the operations outlined in this chapter have implementation-dependent interpretations. Thus, the reader is referred to the CP/M and MP/M file system interface description given in the "PL/I-8Ø Applications Guide" for specific details.

11.1. The READ Statement.

The READ statement is used to read fixed or variable length RECORD SEQUENTIAL files. The form of the READ statement is:

```
READ FILE (fname) INTO (x);
```

If the file is not open, the READ statement provides an automatic OPEN with the attributes RECORD SEQUENTIAL INPUT.

The number of bytes read is determined by the length of x unless the file has been opened with the ENV option which defines the fixed length record. In this latter case, the amount of data read is the declared fixed length and, if the length of x does not match the record size, x is either padded with zeroes or the record is truncated on the right.

11.2. The WRITE Statement.

The WRITE statement transmits data from memory to the data set without conversion. For RECORD SEQUENTIAL files, the format of the WRITE statement is:

```
WRITE FILE (fname) FROM (x);
```

The default OPEN associated with this statement adds the SEQUENTIAL OUTPUT RECORD attributes. Again, the output record size is exactly the length of x, unless the file has been opened with the ENV option, and a fixed length record size was given. In this case, the statement writes the fixed record size, and either pads with zeroes or truncates on the right if the length of x does not match the fixed record length.

11.3. The READ with KEY Statement.

The READ statement with the KEY option is used to directly access individual records within a file. The format of the READ with KEY statement is:

```
READ FILE (fname) INTO (x) KEY (k);
```

where k is a FIXED BINARY expression which defines the relative record to access. Key values start at zero, and continue until the key value times the fixed record length reaches the capacity of the disk. Variable length records cannot be accessed under PL/I-8Ø using the READ with KEY statement.

11.4. The READ with KEYTO Statement.

The READ statement with the KEYTO option allows the key values to be extracted from an input file as it is being sequentially accessed. These key values are normally saved in memory, or in another file, so that records of the input file can later be directly accessed. The form of the READ with KEYTO statement is:

```
READ FILE(fname) INTO(x) KEYTO(k);
```

where k is a FIXED BINARY variable. See the "PL/I Applications Guide" for implementation details.

11.5. The WRITE with KEYFROM Statement.

The WRITE with KEYFROM statement is used to directly access a file for output. The form is:

```
WRITE FILE (fname) FROM (x) KEYFROM (k);
```

where k denotes a FIXED BINARY expression yielding a key value which is treated in the same manner as the READ with KEY option shown above.

12. BUILT-IN FUNCTIONS

A built-in function is a computational subroutine provided as part of the PL/I-8Ø library. A built-in function reference may be used just as a user-defined function reference, except that built-in function names do not have to be declared to be used. If the name of a built-in function is redeclared in the program, the built-in function cannot be called within the scope of that declaration. The built-in function can be used in a contained block, however, by redeclaring it with the attribute BUILTIN. Built-in functions are divided into the following categories, according to their use in PL/I-8Ø.

Arithmetic
Mathemetical
String-handling
Conversion
Condition-handling
Miscellaneous

In the following sections the specific format, parameter attributes, purpose, and properties of each built-in function are described. For a complete listing of all built-in functions see the "PL/I-8Ø Command Summary."

12.1. Arithmetic Functions.

12.1.1. ABS

```
Format:  ABS(X)
Parameters:  X may be any arithmetic expression.
Result:  Returns the absolute value of X.
Algorithm:  If X >= Ø then return X
            otherwise return -X.
Result type:  Same as X.
```

12.1.2. CEIL

```
Format:  CEIL(X)
Parameter:  X is any arithmetic expression.
Result:  Returns the smallest integer greater than
         or equal to X.
Algorithm:  -FLOOR(-X)
Result type:  An integer value of the same type as X.
```

```
12.1.3. DIVIDE
        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 result precision P and scale factor
                 Q, where P and Q are constants, Q assumed
                 to be zero if not included.  Q must be
                 omitted or equal to zero if X and Y are
                 FIXED BINARY.
        Result type: common arithmetic type of X and Y.

12.1.4. FLOOR
        Format:  FLOOR(X)
        Parameter:  X is any arithmetic expression.
        Result:  Computes the greatest integer less than
                 or equal to X.
        Result type:  An integer value of the same type as X.

12.1.5. MAX
        Format:  MAX(X,Y)
        Parameters:  X and Y are arithmetic expressions.
        Result:  Returns the larger value.
        Algorithm:  If X >= Y then return X
                    otherwise return Y.
        Result type:  The common arithmetic type of X and Y.

12.1.6. MIN
        Format:  MIN(X,Y)
        Parameters:  X and Y are arithmetic expressions.
        Result:  Returns the smaller value.
        Algorithm:  If X<= y then return X
                    otherwise return Y.
        Result type:  The common arithmetic type of X and Y.

12.1.7. MOD
        Format:  MOD(X,Y)
        Parameters:  X and Y are arithmetic expressions.
        Result:  Returns the value X mod Y.
        Algorithm:  If Y = Ø then return X
                    otherwise return X-ABS(Y)*FLOOR(X/ABS(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
        Remark:  Note that unless Y = Ø, MOD(X,Y) always
                 returns a non-negative value less than ABS(Y).
```

```
12.1.8. ROUND
        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 >= Ø or -K to the left
                 of the decimal point if K < Ø.
        Algorithm:  Return SIGN(X)*FLOOR(ABS(X)*B**N)+Ø.5)/B**N
                    where B=2 if X is BINARY
                          B=1Ø 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 12345.247ØØ
            ROUND(34567.12345,-3) returns 35ØØØ.ØØØØØ

12.1.9. SIGN
        Format:  SIGN(X)
        Parameter:  X is any arithmetic expression.
        Result:  Returns -1, Ø, or 1 to indicate the sign of X.
        Algorithm:  If X < Ø then return -1
                    If X = Ø then return Ø
                    If X > Ø then return +1
        Result type:  FIXED BINARY

12.1.1Ø. TRUNC
        Format:  TRUNC(X)
        Parameter:  X is any arithmetic expression.
        Result:  Returns the integer portion of X.
        Algorithm:  If X < Ø then return (CEIL(X)).
                    If X >= Ø 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
```

12.2. Mathematical Functions.

The mathematical functions which are provided in the PL/I-8Ø library consist of the most often used trigonometric functions and their inverses, base e (or natural), base 2, and base 1Ø (or common) logarithm functions, the natural exponent function, hyperbolic sin and cos functions, and finally the square root function. Each of these functions is defined for a single FLOAT BINARY argument (other types of arguments are accepted but are automatically converted to this type) and returns a FLOAT BINARY result.

All of the function subroutines, with the exception of SQRT, are based on algorithms which use 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 which can occur using these routines results from the truncation of significant figures during the scaling process. Except for this, the subroutines have an average accuracy of 7.5 significant decimal digits.

Note: For all mathematical functions, the parameter X is assumed to be an arithmetic expression which is converted to FLOAT BINARY and the type of the result is FLOAT BINARY.

12.2.1. ACOS

Format: ACOS(X)
Parameter: X is an arithmetic expression, -1 <= X <= 1.
Result: Returns the arc cosine of X; i.e., ACOS(X) is the angle, in radians, whose cosine is X such that Ø <= 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 ERROR condition is signalled.

12.2.2. ASIN

Format: ASIN(X)
Parameter: X is an arithmetic expression, -1 <= X <= 1.
Result: Returns the arc sine of X; i.e., ASIN(X) is the angle in radians, whose sine is X, and 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 ERROR condition is raised.

12.2.3. ATAN

Format: ATAN(X)
Parameter: X is any arithmetic expression.
Result: Returns the arc tangent of X; i.e., ATAN(X) is the angle in radians, whose tangent is X and such that -PI/2 <= ATAN(X) <= PI/2.
Result type: FLOAT BINARY
Algorithm: Chebyshev polynomial approximation.

12.2.4. ATAND

Format: ATAND(X)
Parameter: X is any arithmetic expression
Result: Returns the arc tangent of X in degrees; i.e., the angle, in degrees, whose tangent is X, and such that -9Ø <= ATAND(X) <= 9Ø.
Result type: FLOAT BINARY
Algorithm: ATAND(X) equals 18Ø/PI * ATAN(X).

12.2.5. COS

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.

12.2.6. COSD

Format: COSD(X)
Parameter: X is an arithmetic expression.
Result: Returns the cosine of X, X in degrees.
Result type: FLOAT BINARY
Algorithm: COSD(X) equals COS(X * PI / 180).

12.2.7. COSH

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.

12.2.8. EXP

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.

12.2.9. LOG

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 ERROR condition is raised.

12.2.10. LOG2

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 ERROR condition is raised.

12.2.11. LOG1Ø
Format: LOG1Ø(X)
Parameter: X is an arithmetic expression, X > Ø.
Result: Returns the logarithm of X to the base 1Ø.
Result type: FLOAT BINARY
Algorithm: LOG1Ø(X) equals LOG(X)/LOG(1Ø).
Error Condition: If X < Ø then the ERROR condition is raised.

12.2.12. SIN
Format: SIN(X)
Parameter: X is an arithmetic expression.
Result: Returns the sine of X, X in radians.
Result type: FLOAT BINARY
Algorithm: Chebyshev polynomial approximation.

12.2.13. SIND
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 / 18Ø).

12.2.14. SINH
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

12.2.15. SQRT
Format: SQRT(X)
Parameter: X is an arithmetic expression, X >= Ø.
Result: Returns the square root of X.
Result type: FLOAT BINARY
Algorithm: Newton's method.
Error Condition: If X < Ø then the ERROR condition is raised.

12.2.16. TAN
Format: TAN(X)
Parameter: X is an arithmetic expression.
Result: Returns the tangent of X, X in radians.
Result type: FLOAT BINARY
Algorithm: If COS(X) = Ø then the ERROR condition otherwise TAN(X) = SIN(X)/COS(X)
Error Condition: If COS(X) equals Ø then the ERROR condition is raised.

12.2.17. TAND

Format: TAND(X)
Parameter: X is an arithmetic expression.
Result: Returns the tangent of X, X in degrees.
Result type: FLOAT BINARY
Algorithm: TAND(X) equals TAN(X * PI / 180)
Error Condition: If COS(X * PI / 180) equals 0 then the ERROR condition is raised.

12.2.18. TANH

Format: TANH(X)
Parameter: X is an arithmetic expression.
Result: Returns the hyperbolic tangent of X.
Result type: FLOAT BINARY
Algorithm: TANH(X) equals (EXP(X)-EXP(-X))/(EXP(X)+EXP(-X))

12.3. String Functions.

```
12.3.1. BOOL
        Format:  BOOL(X,Y,Z)
        Parameters:  X is a bit expression.
                     Y is a bit expression.
                     Z is a bit string constant,
                       4 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    BØØL(A,B,Z)
                       ----------------------
                       | Ø    Ø        Z1     |
                       |                      |
                       | Ø    1        Z2     |
                       |                      |
                       | 1    Ø        Z3     |
                       |                      |
                       | 1    1        Z4     |
                       |----------------------|

                 This then induces the function BOOL(X,Y)
                 on bit strings X,Y as follows:  If X,Y do
                 not have the same length then the shorter
                 string is padded on the right with zeroes
                 until they have the same length.  Then
                 BØØL(X,Y,Z) is defined to be the bit string
                 whose Nth bit is obtained from the above
                 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('ØØ11'B,'Ø1Ø1'B,'1ØØ1'B) returns '1ØØ1'B
           BOOL('Ø1Ø11'B,'11','1ØØ1') returns 'Ø11ØØ'


12.3.2. COLLATE
        Format:  COLLATE()
        Parameters:  None
        Result:  Returns a character string of length 128
                 that consists of the set of characters in
                 the ASCII character set in ascending order.
                 The ASCII character set is given in
                 Appendix A.
        Result type:  CHARACTER(128)
```

12.3.3. INDEX

```
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, the
         value returned is zero.
Result type:  FIXED BINARY
```

12.3.4. LENGTH

```
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
```

12.3.5. SUBSTR

```
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 which is a copy of
         the string S beginning at the Ith
         element and for a length J.  If J
         is not given it is assumed to be
         equal to LENGTH(X)-I+1.
Result type:  Same as X.
Error Condition:  None.  If the parameters
                  are out of range, unpredictable
                  results will be obtained.
```

12.3.6. TRANSLATE

```
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.  Then, any
         occurrence of a character in Z in
         the string X is replaced by the
         character in Y corresponding to the
         character in Z.
Result type:  Same as X.
Examples:
    TRANSLATE('BDA','1234','ABC') returns '2D1'
```

```
12.3.7. VERIFY
        Format:  VERIFY(X,Y)
        Parameters:  X is a character expression.
                     Y is a character expression.
        Result:  Returns integer value Ø if each of
                 the characters in X occurs in Y.
                 Otherwise returns an integer which
                 indicates the leftmost character of
                 X which does not occur in Y.
        Result type:  FIXED BINARY
        Examples:
                 VERIFY('ABCDE7,7ABDE7) returns 3
                 VERIFY('ABC123','1A2B3C4D') returns Ø.
                 VERIFY('','A') returns Ø.
                 VERIFY('A','') returns 1.
```

12.4. Conversion Functions.

These functions allow the user to convert one type of data item to another type and are used internally for automatic type conversions.

```
12.4.1. ASCII
        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.
                 For ASCII codes, see Appendix A.
        Result type:  CHARACTER(1)
        Algorithm:  ASCII(X) equals
                    SUBSTR(COLLATE(),MOD(X,128)+1),1)
        Remark:  ASCII(X) is the inverse function to RANK(X).

12.4.2. BINARY
        Format:  BINARY(X[,P])
        Parameter:  X is an arithmetic expression, or string
                    expression which can be converted to an
                    arithmetic value.  If X is DECIMAL with a
                    non-zero scale factor, then P must be given,
                    where P is an integer constant which 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.
```

```
12.4.3. BIT
        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, converts S to a
                 bit string whose length is determined by the
                 conversion rules in Chapter 7.
        Result type:  BIT

12.4.4. CHARACTER
        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 Chapter 7.
        Result type:  CHARACTER

12.4.5. DECIMAL
        Format:  DECIMAL(X[,P[,K]])
        Parameter:  X is an arithmetic or string expression which
                    can be converted to an arithmetic value.
                    P is an integer constant, 1 <= P <= 15.
                    K is an integer constant, Ø <= K <= P.
        Result:  Converts X to a DECIMAL value.  P and K are
                 optional but when specified represent the
                 precision and scale factor, respectively.
                 If only P is given, K is assumed to be zero.
                 If neither P nor K is given, then the precision
                 and scale factor of the result are determined
                 by the rules for conversion given in Chapter 7.
        Result type:  FIXED DECIMAL

12.4.6. DIVIDE
        Format:  DIVIDE(X,Y,P[,Q])
        Parameters:  X and Y are arithmetic expressions.
                     P,Q are integer constants, Ø <= Q <= P.
        Result:  Returns X divided by Y with precision P
                 and scale Q.  Q is optional but must not
                 be given if X and Y are both FIXED BINARY.
        Result type:  The common type of X and Y.
```

```
12.4.7. FIXED
        Format:  FIXED(X[P,[,K]])
        Parameters:  X is an arithmetic expression or string
                     expression which can be converted to an
                     arithmetic value.
                     P is an integer constant,
                     K is an integer constant.
        Result:  Converts X to a FIXED arithmetic value.  P and K
                 are optional but when specified determine the
                 precision and scale factor, respectively, of the
                 result.  If only P is given, then K is assumed
                 to be zero.  If neither P nor K is given, then
                 the precision and scale factor are determined
                 by the conversion rules in Chapter 7.
        Result type:  If X is FIXED DECIMAL or CHARACTER, the
                      result is FIXED DECIMAL; otherwise, it
                      is FIXED BINARY.

12.4.8. FLOAT
        Format:  FLOAT(X[,P])
        Parameter:  X is an arithmetic  or string expression
                    which 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
                 Chapter 7.
        Result type:  FLOAT BINARY

12.4.9. RANK
        Format:  RANK(X)
        Parameter:  X is a character value of length one.
        Result:  Returns the integer representation of the ASCII
                 character X.  See Appendix A.
        Result type:  FIXED BINARY
        Algorithm:  RANK(X) equals INDEX(COLLATE(),X) -1

12.4.10. UNSPEC
         Format:  UNSPEC(X)
         Paarameter:  X is a reference to a data item whose
                      internal representation in memory is
                      16 bits or less.
         Result:  Returns the contents of the address referenced
                  by X.
         Result type:  A bit string whose length equals the length
                       of the internal representation of the data
                       item associated to X.
```

12.5. Condition Functions.

These functions allow the PL/I-8Ø user to investigate interrupts caused by enabled conditions. None of these functions have parameters, and they return a vlue only when executed in an ON-unit that is entered as a result of an interrupt caused by one of the conditions for which the function can be used, or when such a condition is signalled.

12.5.1. ONCODE

Format: ONCODE()
Result: The vlue returned is the error number of the most recent PL/I-8Ø runtime error which signalled the ERROR condition. The error messages and their corresponding error numbers are listed in Appendix F.
Result type: FIXED BINARY

12.5.2. ONFILE

Format: ONFILE()
Result: The value returned is the file name for which the most recent ENDFILE or ENDPAGE condition was signalled.
Result type: CHARACTER

12.5.3. ONKEY

Format: ONKEY()
Result: The value returned is a character string giving the value of the key for the record that caused an input/output or conversion condition to be raised.

12.6. Miscellaneous Functions.

12.6.1. ADDR

Format: ADDR(X)
Parameter: X is a reference to a variable whose storage is connected.
Result: Returns a pointer that identifies the location to which the variable X has been allocated.
Result type: POINTER

12.6.2. DIMENSION
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

12.6.3. HBOUND
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

12.6.4. LBOUND
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

12.6.5. LINENO
Format: LINENO(X)
Parameter: X is a file value.
Result: Returns the linenumber of the file control block referenced by X. This file control block must have the PRINT attribute.
Result type: FIXED BINARY

12.6.6. NULL
Format: NULL
Result: Returns a null pointer value; i.e., a pointer which points to no location.
Result type: POINTER

12.6.7. PAGENO
Format: PAGENO(X)
Parameter: X is a file value.
Result: Returns the pagenumber of the file control block referenced by X. The file control block must have the PRINT attribute.
Result type: FIXED BINARY

APPENDIX A

TABLE OF ASCII CODES AND ESCAPE CHARACTERS

	N	HEX	ASCII	N	HEX	ASCII	N	HEX	ASCII
^@	Ø	ØØ	NUL	43	2B	+	86	56	V
^A	1	Ø1	SOH	44	2C	,	87	57	W
^B	2	Ø2	STX	45	2D	-	88	58	X
^C	3	Ø3	ETX	46	2E	.	89	59	Y
^D	4	Ø4	EOT	47	2F	/	9Ø	5A	Z
^E	5	Ø5	ENQ	48	3Ø	Ø	91	5B	[
^F	6	Ø6	ACK	49	31	1	92	5C	\
^G	7	Ø7	BEL	5Ø	32	2	93	5D	]
^H	8	Ø8	BS	51	33	3	94	5E	^
^I	9	Ø9	HT	52	34	4	95	5F	
^J	1Ø	ØA	LF	53	35	5	96	6Ø	T
^K	11	ØB	VT	54	36	6	97	61	a
^L	12	ØC	FF	55	37	7	98	62	b
^M	13	ØD	CR	56	38	8	99	63	c
^N	14	ØE	SO	57	39	9	1ØØ	64	d
^O	15	ØF	SI	58	3A	:	1Ø1	65	e
^P	16	1Ø	DLE	59	3B	;	1Ø2	66	f
^Q	17	11	DC1	6Ø	3C	<	1Ø3	67	g
^R	18	12	DC2	61	3D	=	1Ø4	68	h
^S	19	13	DC3	62	3E	>	1Ø5	69	i
^T	2Ø	14	DC4	63	3F	?	1Ø6	6A	j
^U	21	15	NAK	64	4Ø	@	1Ø7	6B	k
^V	22	16	SYN	65	41	A	1Ø8	6C	l
^W	23	17	ETB	66	42	B	1Ø9	6D	m
^X	24	18	CAN	67	43	C	11Ø	6E	n
^Y	25	19	EM	68	44	D	111	6F	o
^Z	26	1A	SUB	69	45	E	112	7Ø	p
^[	27	1B	ESC	7Ø	46	F	113	71	q
^\	28	1C	FS	71	47	G	114	72	r
^]	29	1D	GS	72	48	H	115	73	s
^~	3Ø	1E	RS	73	49	I	116	74	t
^_	31	1F	US	74	4A	J	117	75	u
	32	2Ø	SP	75	4B	K	118	76	v
	33	21	!	76	4C	L	119	77	w
	34	22	"	77	4D	M	12Ø	78	x
	35	23	#	78	4E	N	121	79	y
	36	24	$	79	4F	O	122	7A	z
	37	25	%	8Ø	5Ø	P	123	7B	{
	38	26	&	81	51	Q	124	7C	|
	39	27	'	82	52	R	125	7D	}
	4Ø	28	(	83	53	S	126	7E	~
	41	29	)	84	54	T	127	7F	DEL

APPENDIX B

PL/I-8Ø STATEMENTS

The PL/I-8Ø statement formats are listed below in alphabetical order followed by their corresponding section numbers in parentheses.

All statements may have label prefixes. The label prefixes are omitted in the formats except in the statements that require them.

B.1 THE ALLOCATE STATEMENT (6.5)

```
ALLOCATE based-variable SET(pointer-variable);
```

Example:

```
    DCL A CHAR(16) BASED(P),
        P POINTER;
    ALLOCATE A SET(P);
```

B.2 THE ASSIGNMENT STATEMENT (7)

```
variable = expression;
```

Examples:

```
    B = C*D;
    UNSPEC(E) = F(I);
```

B.3 THE BEGIN STATEMENT (2.3)

```
BEGIN;
```

B.4 THE CALL STATEMENT (8.11)

```
CALL procname[(arg1, ...,argN)];
```

Examples:

```
     CALL P1;
     CALL P2(A,B,C);
```

B.5 THE CLOSE STATEMENT (9.6)

```
CLOSE FILE(fname);
```

Examples:

```
     CLOSE FILE(INP);
     CLOSE FILE(OUT);
```

B.6 THE DECLARE STATEMENT (5.1)

```
DECLARE|DCL [level] name [attribute-list] ...

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

Examples:

```
DCL A FIXED;
DCL 1 B,
      2 C NAME CHAR(2Ø),
      2 D ADDRESS,
        3 STREET CHAR(2Ø),
        3 CITYST CHAR(2Ø),
        3 ZIP CHAR(5);
DCL ZZ(1Ø) FIXED;
DCL A FIXED EXTERNAL;
```

B.7 THE DO STATEMENT (2.2, 8.3)

```
DO [control-var] spec;
```

where spec may be one of the following:

```
[start-exp [TO end-exp] [BY incr-exp]] [WHILE(cond)]

[start-exp [BY incr-exp] [TO end-exp]] [WHILE(cond)]

[start-exp [REPEAT(repeat-exp)]] [WHILE(cond)]
```

Examples:

```
DO J=Ø;
DO WHILE(A<B);
DO J = 1 TO 1Ø;
DO K = 1Ø TO Ø BY -2 WHILE(A<B);
DO P=START REPEAT P->NEXT WHILE(P~=NULL);
```

B.8 THE END STATEMENT (2.2, 2.3, 8.12)

```
END [label];
```

Examples:

```
END;
END P1;
```

B.9 THE FORMAT STATEMENT (1Ø.9)

```
label: FORMAT(format-list);
```

Examples:

```
L1: FORMAT(A(5));
L2: FORMAT(1ØB4(2));
```

B.10 THE FREE STATEMENT (6.3)

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

Examples:

```
     FREE A;
     FREE P->A;
```

B.11 THE GET EDIT STATEMENT (10.10)

```
GET [FILE(fname)] [SKIP[(nl)]] [EDIT(input-list)
                                    (format-list)];
```

Examples:

```
     GET EDIT(A,B,C)(3F(5,2));
     GET FILE(INP) EDIT((Z(I) DO I = 1 TO 3))(A);
```

B.12 THE GET LIST STATEMENT (10.2)

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

Example:

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

B.13 THE GOTO STATEMENT (8.1)

```
GOTO|GO TO label-constant|label-variable;
```

Examples:

```
     GO TO THEEND;
     GO TO LAB(K);
```

B.14 THE IF STATEMENT (8.2)

```
IF cond THEN action [ELSE [action2]];
```

Examples:

```
     IF A=2 THEN B=A**2;
     ELSE;

     IF J>K THEN I = I+1;
     ELSE I = I+3;
```

B.15 THE EMPTY STATEMENT (8.2)

```
;
```

Examples:

```
     ;
     ELSE ;
```

B.16 THE ON STATEMENT (8.5)

```
ON condition ON-unit

Examples:
      ON ENDFILE(INP)
          BEGIN;
          PUT LIST('END OF INPUT');
          STOP;
          END;
       ON ERROR PUT LIST(ONCODE());
```

B.17 THE OPEN STATEMENT (9.3)

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

Examples:
      OPEN FILE(INP) INPUT;
      OPEN FILE(SYSPRINT) OUTPUT;
```

B.18 THE PROCEDURE STATEMENT (8.12)

```
procname:  PROCEDURE|PROC [(parml, ...,parmN)]
    [OPTIONS(MAIN)] [RETURNS(attribute-list)] [RECURSIVE]

Examples:
Pl:  PROC(A,B,C);
P2:  PROCEDURE (ZZ) RETURNS(FLOAT);
P3:  PROC(N) RETURNS(FIXED BIN) RECURSIVE;
P4:  PROCEDURE OPTIONS(MAIN);
```

B.19 THE PUT EDIT STATEMENT (10.11)

```
PUT [FILE(fname)] [SKIP[nl]] [PAGE[(p)]]
    [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);
```

B.20 THE PUT LIST STATEMENT (10.3)

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

Examples:
     PUT LIST(A,B,C);
     PUT FILE(F) LIST((Z(I) DO I = 1 TO 10));
```

B.21 THE READ STATEMENT (For STREAM files) (10.13)

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

Examples:

```
     DCL (VV,S) CHAR(200) VAR;
     READ INTO(VV);
     READ FILE(INP) INTO(S);
```

B.22 THE READ STATEMENT (SEQUENTIAL RECORD) (11.1)

```
READ FILE(fname) INTO (x);
```

Example:

```
     READ FILE(INP) INTO (XX);
```

B.23 THE READ STATEMENT (with KEYTO) (11.4)

```
READ FILE(fname) INTO(x) KEYTO(keyto);
```

Example:

```
     READ FILE(INP) INTO(Z) KEYTO(IKEY);
```

B.24 THE READ STATEMENT (with KEY) (11.3)

```
READ FILE(fname) INTO (x) KEY(ikey);
```

Example:

```
     READ FILE(INP) INTO(STRUC) KEY(IKEY);
```

B.25 THE RETURN STATEMENT (8.13)

```
RETURN [(exp)];
```

Examples:

```
     RETURN;
     RETURN(X);
     RETURN(A**2);
```

B.26 THE REVERT STATEMENT (8.7)

```
REVERT condition;
```

Examples:

```
     REVERT ERROR;
     REVERT ENDFILE;
```

B.27 THE SIGNAL STATEMENT (8.6)

```
SIGNAL condition;

Examples:
     SIGNAL ERROR;
     SIGNAL ENDFILE;
```

B.28 THE STOP STATEMENT (8.15)

```
STOP;
```

B.29 THE WRITE STATEMENT (with STREAM files) (10.14)

```
WRITE [FILE(fname)] FROM(v);

Example:
     DCL (XX,YY) CHAR(200) VAR;
     WRITE FILE(OUTPUT) FROM(XX);
     WRITE FROM(YY);
```

B.30 THE WRITE STATEMENT (SEQUENTIAL RECORD) (11.2)

```
WRITE FILE(fname) FROM(x);

Examples:
     WRITE FILE(OUTP) FROM (XX);
     WRITE FILE(F) FROM(STRUC);
```

B.31 THE WRITE STATEMENT (with KEYFROM) (11.5)

```
WRITE FILE(fname) FROM(x) KEYFROM(ikey);

Example:
     WRITE FILE(KP) FROM(REC) KEYFROM(IKEY);
```

APPENDIX C

DATA ATTRIBUTES

The PL/I-8Ø data attributes are listed below in alphabetical order followed by their corresponding section numbers in parentheses.

C.1 ALIGNED (6.2)

```
Example:
    DCL A(Ø:3) BIT (4) ALIGNED;
```

C.2 AUTOMATIC | AUTO (6.3)

```
Example:
    DCL A FIXED BIN;  is equivalent to
    DCL A FIXED BIN AUTO;
```

C.3 BASED [(p)] (6.4)

```
Example:
    DCL A CHAR(1Ø) BASED(P),
        B(5) FIXED BIN BASED;
```

C.4 BINARY | BIN (3.1.1, 5.2)

```
Example:
    DCL I FIXED BIN,
        F FLOAT BIN;
```

C.5 BIT[(n)] (3.2.2)

```
Example:
    DCL A BIT(3);
```

C.6 BUILTIN (5.2,12)

```
Example:
    DCL SQRT BUILTIN;
```

C.7 CHARACTER | CHAR [(n)] (3.2.1,5.2)

```
Example:
    DCL A CHAR(1Ø),
        B(5) CHAR(4);
```

C.8 DECIMAL | DEC (3.1.2,5.2)

```
Example:
    DCL A FIXED DEC(6,2);
```

C.9 ENTRY[(att1, ...,attN)] (3.3.2, 9.4.1, 5.2)

Example:
```
    DCL H ENTRY,
        Z ENTRY ((10) FIXED),
        Y ENTRY (FLOAT) RETURNS (FLOAT),
        X ENTRY VARIABLE;
```

C.10 EXTERNAL | EXT (5.1, 5.2)

Example:
```
    DCL A CHAR(8) EXTERNAL;
```

C.11 FILE (3.5, 5.2)

Example:
```
    DCL F FILE,
        FV FILE VARIABLE;
```

C.12 FIXED (3.1.1, 3.1.2)

Example:
```
    DCL A FIXED BIN,
        B FIXED DECIMAL(5,2);
```

C.13 FLOAT (3.1.3)

Example:
```
    DCL A FLOAT BIN;
```

C.14 INITIAL | INIT (value [,value]...) (6.2)

Example:
```
    DCL A CHAR(3) STATIC INIT('ABC'),
        B(2) FIXED BIN STATIC INIT(2(5));
```

C.15 LABEL (3.1.1)

Example:
```
    DCL WHERE LABEL;
```

c.16 POINTER | PTR (3.4, 6.4)

Example:
```
    DCL (P,Q) POINTER;
```

C.17 RETURNS (attribute-list) (8.17)

Example:
```
    DCL A ENTRY(FLOAT) RETURNS(FIXED);
```

C.18 STATIC (6.1)

Example:
```
    DCL A CHAR(10) STATIC,
        B FIXED BIN STATIC INIT(0);
```

C.19 VARIABLE (3.3.2, 3.5, 9.1)

Example:
```
    DCL F FILE VARIABLE,
        P ENTRY VARIABLE;
```

C.20 VARYING | VAR (3.2.1)

Example:
```
    DCL A CHAR(100) VAR;
```

APPENDIX D

PICTURE FORMAT ITEM

This appendix describes an added feature, the PICTURE output data format item, which is implemented in PL/I-80 beginning with Version 1.3. This feature is implemented in conformity with the ANSI Committee X3J PL/I Subset G Standard together with the full ANSI PL/I Standard.

D.0 Picture Syntax

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 occurring in the PICTURE format item. The syntax of a PICTURE format item is:

P<picspec>

where <picspec> is a character string constant describing the picture specification. Such a format item may be used in a PUT EDIT statement in the same manner as any other data format item (see Section 10.6). The character string constant used to describe the picture specification must consist of one or more of the following characters:

$ + - 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

and must satisfy certain rules of syntax. First, insertion characters may occur anywhere in a valid <picspec>, except they may not separate the characters of either of the picture character pairs, CR and DB. If all insertion characters of a picture specification are removed, the resulting string must be acceptable to the (non-deterministic) finite state machine recognizer which is illustrated in Figure D.1. 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 unlabelled, or if the edge is labeled by the next character(s) in the <picspec>.

For example, the following character string constants define valid picture specifications:

```
'BB$***,***V.99BB'

'$----,999V.99BCR'

'99:99:99'

'**/**/**'

':BBB$SSSS,SSS.VSSBBB:'
```

D.1 Picture Semantics

The manner in which a picture specification edits a numeric value into a character string value is determined by the types of picture characters appearing in the specification. The characters '$','+', '-',and 'S' occur as either 'static' characters, or 'drifting characters'. Such a character is static if it appears only once in the picture specification; 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, will determine an output character, given by the following table, which will occupy one position in the output.

	STATIC/DRIFTING CHARACTERS			
SIGN	S	+	-	$
pos	+	+	' '	$
neg	-	' '	-	$

If the picture character is static, the output character will appear in the corresponding position of the output. If the picture character is drifting, then the output character will appear exactly one position ahead of the first non-zero digit over which the picture character drifts, or in the last position over which it drifts. All other occurrences of the drifting character will be replaced by spaces, corresponding to the suppression of a zero digit in the numeric value.

The characters '*' and 'Z' are called 'conditional digit' picture characters or 'zero suppression' characters. Each such character in the picture specification is associated with a digit in the numeric value. On output, if the corresponding digit is a zero, the output character will be a '*' or ' ', respectively. If the corresponding digit is non-zero, the output will be the digit character.

The picture characters 'B','/','.',':',and ',' are called insertion characters' (the character ':' is not an insertion character defined in the ANSI Standard but has been added in PL/I-80 for the purpose of displaying numeric data which represents time). Insertion characters result in that character occurring in the corresponding output position (B results in a space), unless the insertion character occurs within 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 which is causing the suppression of numeric digits, the insertion character will also be suppressed following the rules above.

Note: In some implementations, the 'B' is an unconditional insertion character, i.e. it always causes a space in the corresponding position of the output. However, by the ANSI Standard, such a space in the output can be overwritten by a drifting character or the zero suppression character '*'.

The picture character '9' in a picture specification specifies that the corresponding digit in the numeric value will occur in the corresponding position of the output. Thus '9' is an unconditional digit position.

The correspondence between digits in the numeric value and the numeric digit positions in the picture specification is established by the 'V' picture character. This character serves only to specify the position where integral digits end and fractional digits begin, and thus specifies the alignment of the picture specification to the numeric value. If this character does not occur, it is assumed that all the digit positions implied by the picture specification refer to integral digit positions (no fractional) and thus any fractional digits in the numeric value will not appear in the result. Note that the 'V' picture character is the only character which does not correspond to a character position in the result. Thus the length of the result equals the length of the picture specification if 'V' does not appear, but is one character less if 'V' appears. The 'V' character also has an effect on suppression of characters. The fractional digits, i.e. digits corresponding to positions past the 'V', are never suppressed unless all of them are suppressed (in which case everything is suppressed). Generally, beyond the 'V', suppression is turned OFF if it is ON. As a result an insertion character which occurs beyond the 'V' picture character, for example a decimal point, is not suppressed unless everything is suppressed.

The character pairs 'CR' and 'DB', representing 'credit' and 'debit', are viewed as sign characters. If either of these appear in the picture specification, and if the sign of the numeric value is negative, then the specified pair will occur in the result. If the numeric value is positive, then the positions corresponding to these character pairs will be replaced by two spaces.

In addition to the above rules, there are some general rules for default cases. If the numeric value is zero, and if the picture specification does not contain a ´9´ picture character, then the resulting output will be all *´s if the picture character ´*´ occurs at all, and all spaces otherwise. This rule takes precedence over the above rules. Also, if the sign of the numeric value is negative, and if none of the sign picture characters S,+,-,CR, or DB occur in the specification, then ERROR(1), a conversion error, is signalled.

Each picture specification 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. Except for this, the precision of the result will equal 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 will be 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.

Figure D.2 illustrates some of the rules involving the use of PICTURE data format items.

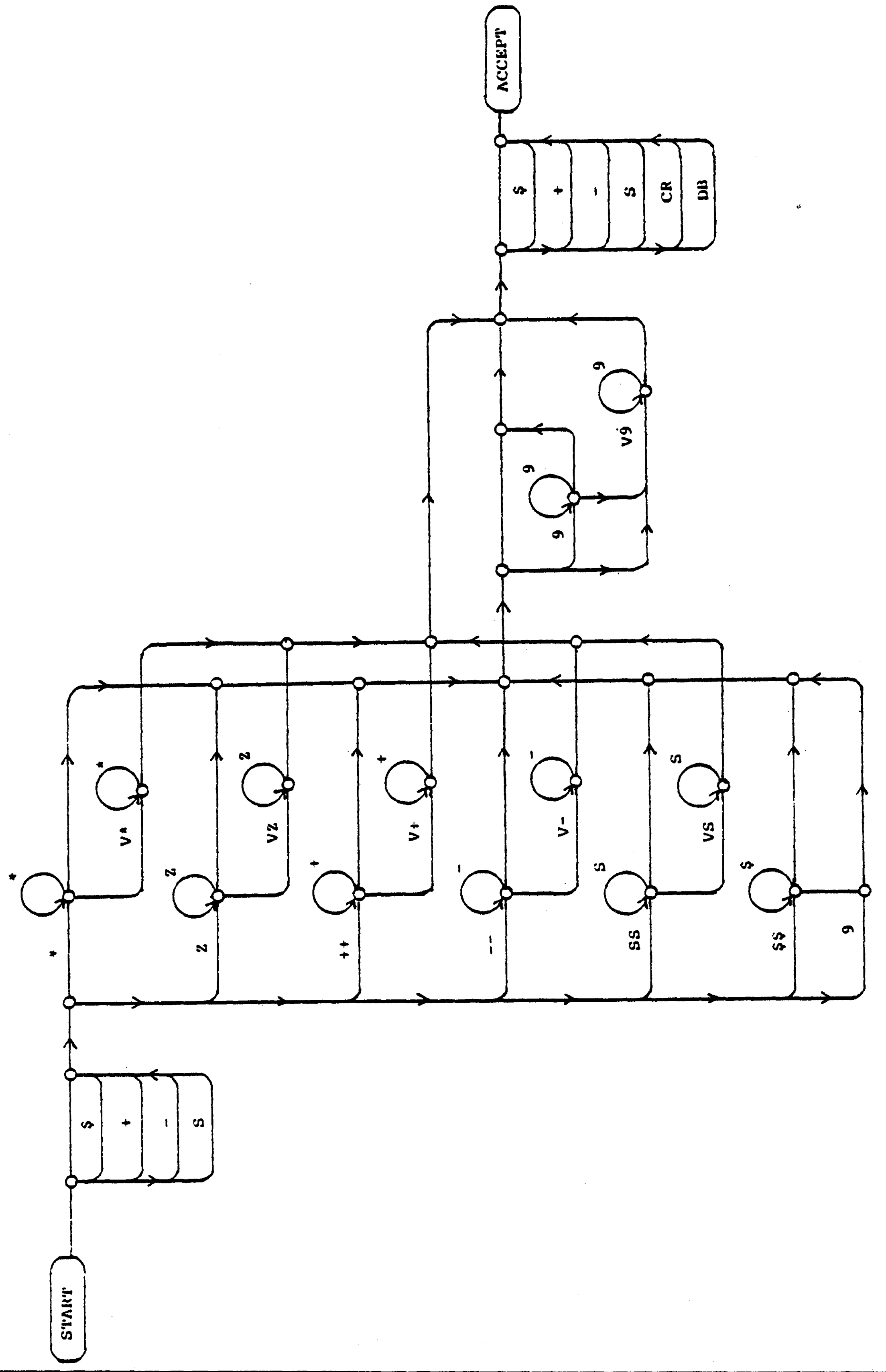

FIGURE D.1 PICTURE SPECIFICATION RECOGNIZER

```
     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     $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     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/
```

FIGURE D.2 EXAMPLES OF PICTURE EDITED NUMERIC DATA

```
     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


     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/
```

APPENDIX E

EXTERNAL PROCEDURES

This appendix describes the use of the EXTERNAL attribute as applied to local procedure definitions, available in PL/I-80 version 1.3 or later. This is a non-standard feature, and must be avoided if upward compatibility with other Subset-G implementations is required.

It is often useful to group a set of separately compiled procedures into a single compilation, where the procedures reference the same global data. According to the Subset-G standard, each subroutine must be separately compiled, and the global data must be duplicated in each compilation. The individual modules are then combined using the linkage editor to produce the final object module. The EXTERNAL attribute can be applied to a procedure heading in PL/I-80 in order to make the procedure accessible outside the module. In order to be compatible with any future implementations of PL/I from Digital Research, only top-level procedures should be marked with the EXTERNAL attribute, and all globally-accessed data should be marked as STATIC. A compilation containing a group of EXTERNAL subroutines should consist only of subroutines, and no main program. The following program segment shows the use of the EXTERNAL attribute:

```
module:
    proc;
    dcl
        1 global_data static,
          2 a_field char(20) var init(''),
          2 b_field fixed       init(0),
          2 c_field float       init(0);
    set_a:
        proc (c) external;
      dcl c char(20) var;
          a_field = c;
        end set_a;
    set_b:
        proc (x) external;
        dcl x fixed;
        b_field = x;
        end set_b;
    set_c:
        proc (y) external;
        dcl y float;
        c_field = y;
        end set_c;
    sum:
        proc returns(float) external;
        return (b_field+c_field);
        end sum;
    display:
        proc external;
        put skip list(a_field,b_field,c_field);
        end;
    end module;
```

The program shown above defines five external procedures: set_a, set_b, set_c, sum, and display. These four procedures are accessed in the program shown below:

```
        call_ext:
            proc options(main);
            dcl
                set_a entry (char(20) var),
                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 programs are separately compiled and linked together to form a single module. Note that, due to linkage editor format restrictions, long external names are truncated on the right and thus must be unique in the first six characters.