Form Y33-6003-0

Language Specifications

IBM System/360

PL/I Language Specifications

This publication is a description of the PL/I language. It does not describe any implementation; nor can it be construed that the publication implies any commitment that the features are implemented or will be implemented by IBM. The publication is intended for the use of implementers and programming language designers concerned with language development and the study of languages.

PREFACE

This publication is a specifications manual for the entire PL/I language. It is not intended to reflect any implementation. The book is designed for the use of implementers, systems programmers, and others who need to know the language beyond the present implementations. The following books describe the language as implemented for the F-level compiler and the D-level compiler:

> IBM System/360: PL/I Reference Manual, Form C28-8201
>
> IBM System/360: PL/I Subset Reference Manual, Form C28-8202

There are other IBM publications that perform a tutorial function. These publications and their intended audience are as follows:

A PL/I Primer, Student Text, Form C28-6808, is intended for the novice programmer who has little or no knowledge of data processing, as well as for the experienced programmer who wants to learn PL/I.

A Guide to PL/I for FORTRAN Users, Student Text, Form C20-1637, is directed toward the programmer who has a working knowledge of FORTRAN.

A Guide to PL/I for Commercial Programmers, Student Text, Form C20-1651, is intended for the programmer who has experience in commercial applications. Comparisons between PL/I and COBOL (COmmon Business Oriented Language) are included in this guide.

Major Revision (March 1968)

This publication, Form Y33-6003-0, makes obsolete the previous edition, Form C28-6571-4.

In addition to language changes and new language features, much of the material has been rewritten and reordered. Changes and additions to the language are identified by vertical bars to the left of the text. Language changes that might affect existing programs are:

1. Priority of the concatenation operator
2. Conversion of character-string and bit-string data to arithmetic
3. ALIGNED and PACKED attributes, including change of keyword PACKED to UNALIGNED
4. The form of the AREA attribute
5. The defaults of REDUCIBLE and IRREDUCIBLE
6. Rounding of E and F format items

FIGURES

PRESENTATION OF INFORMATION IN THIS MANUAL

It is not intended that this manual should be read sequentially. It is essentially a reference book and an understanding of the information in it does not depend on having read preceding information.

The index should be consulted whenever seeking information to support statements in the text.

SYNTAX NOTATION IN THIS MANUAL

Throughout this manual, wherever a PL/I statement -- or some other combination of elements -- is discussed, the manner of writing that statement or phrase is illustrated with a uniform system of notation.

This notation is not a part of PL/I; it is a standardized notation that may be used to describe the syntax -- or construction -- of any programming language. It provides a brief but precise explanation of the general patterns that the language permits. It does not describe the meaning of the language elements, merely their structure; that is, it indicates the order in which the elements may (or must) appear, punctuation that is required, and options that are allowed.

The following rules explain the use of this notation for any programming language; only the examples apply specifically to PL/I:

1. A notation variable is the name of a general class of elements in the programming language. A notation variable must consist of:

   a. Lower-case letters, decimal digits, and hyphens and must begin with a letter.

   b. A combination of lower-case and upper-case letters. There must be one portion in all lower-case letters and one portion in all upper-case letters, and the two portions must be separated by a hyphen.

   All such variables used are defined in the manual either formally, using this notation, or are defined in prose.

Examples:

   a. digit. This denotes the occurrence of a digit, which may be 0 through 9 inclusive.

   b. filename. This denotes the occurrence of the notation variable named filename. An explanation of filename is given elsewhere in the manual.

   c. DO-statement. This denotes the occurrence of a DO statement. The upper-case letters are used for emphasis.

2. A notation constant denotes the literal occurrence of the characters represented. A notation constant consists either of all CAPITAL letters or of a special character.

Example:

```
DECLARE identifier FIXED;
```

This denotes the literal occurrence of the word DECLARE followed by the notation variable "identifier," which is defined elsewhere, followed by the literal occurrence of the word FIXED followed by the literal occurrence of the semicolon (;).

3. The term "syntactical unit," which is used in subsequent rules, is defined as one of the following:

   a. a single variable or constant, or

   b. any collection of variables, constants, syntax-language symbols, and reserved words surrounded by braces or brackets.

4. Braces { } are used to denote grouping.

Example:

$$\text{identifier} \begin{Bmatrix} \text{FIXED} \\ \text{FLOAT} \end{Bmatrix}$$

The vertical stacking of syntactical units indicates that a choice is to be made. The above example indicates that the variable "identifier" must be followed by the literal occurrence of either the word FIXED or the word FLOAT.

5. The vertical stroke | indicates that a choice is to be made.

Example:

```
identifier {FIXED|FLOAT}
```

This has exactly the same meaning as the above example. Both methods are used in this manual to display alternatives.

6. Square brackets [ ] denote options. Anything enclosed in brackets may appear one time or may not appear at all.

Example:

```
CHARACTER (length) [VARYING]
```

This denotes the literal occurrence of the word CHARACTER followed by the notation variable "length" enclosed in parentheses and optionally followed by the literal occurrence of the word VARYING. If, in rule 4, the two alternatives also were optional, the vertical stacking would be within brackets, and there would be no need for braces.

7. Three dots ... denote the occurrence of the immediately preceding syntactical unit one or more times in succession.

Example:

```
[digit] ...
```

The notation variable, "digit," may or may not occur since it is surrounded by brackets. If it does occur, it may be repeated one or more times.

8. Underlining is used to denote an element in the language being described when there is conflict between this element and one in the syntax language.

Example:

operand {&|<u>|</u>} operand

This denotes that the variables "operand" are separated by either an "and" (&) or an "or" (|). The notation constant | is underlined in order to distinguish the "or" symbol in the PL/I language from the "or" symbols in the syntax language.

BASIC LANGUAGE STRUCTURE

PL/I allows the programmer to write the statements of his program in a free-field format. A statement, which is a string of characters, is always terminated by the special character, semicolon. A program which is, in turn, a sequence of statements, can thus be regarded simply as a single string of characters, with no special internal grouping. Hence, a PL/I program can be physically represented and transmitted to a computer in a natural way by means of almost any input medium, including a typewriter at a remote terminal.

Input conventions, depending upon the machine configuration or the compiler, can, of course, be set up so that the program string may be presented to the computer through the familiar medium of fixed-length records, e.g., punched cards. This can be accomplished by using certain predetermined fields of the records for the program string, and other fields for arbitrary purposes.

LANGUAGE CHARACTER SETS

One of two character sets may be used to write a source program: either a 60-character set or a 48-character set. No assumptions are made in the language about external or internal codes for the characters. For a given program, the choice between the two sets is optional. (In practice, this choice will depend upon the available equipment.)

60-Character Set

The 60-character set is composed of digits, special characters, and English language alphabetic characters.

There are 29 alphabetic characters, letters A through Z and three additional characters (alphabetic extenders) that are defined as and treated as alphabetic characters. These characters and the graphics by which they are represented are as follows:

Currency symbol	$
Commercial At-sign	@
Number sign	#

There are ten digits. Decimal digits are the digits 0 through 9. A binary digit (bit) is either a 0 or a 1.

An alphameric character is either an alphabetic character or a digit.

There are 21 special characters. The names and graphics by which they are represented are:

Name	Graphic
Blank	
Equal or Assignment symbol	=
Plus	+
Minus	-
Asterisk or Multiply symbol	*
Slash or Divide symbol	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point or Period	.
Quotation mark	'
Percent symbol	%
Semicolon	;
Colon	:
Not symbol	¬
And symbol	&
Or symbol	|
Greater Than symbol	>
Less Than symbol	<
Break_character (used as shown)	_
Question mark	?

Note that the quotation mark used in PL/I is the single quotation mark (also known as an apostrophe or prime).

Two consecutive special characters are sometimes used as operators, e.g., >=, denoting "greater than or equal to"; ||, denoting concatenation.

48-Character Set

The characters making up the 48-character set are identical to those of the 60-character set, with restrictions and changes as described in Appendix 5.

DELIMITERS

Certain characters are used as <u>delimiters</u> and fall into three classes:

- operators
- parentheses
- separators and other delimiters

Operators

<u>Operators</u> used by the language are divided into four types:

- arithmetic operators
- comparison operators
- bit-string operators
- string operators

Arithmetic Operators

The <u>arithmetic operators</u> are:

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

Comparison Operators

The <u>comparison operators</u> are:

>	denoting greater than
¬>	denoting not greater than
>=	denoting greater than or equal to
=	denoting equal to
¬=	denoting not equal to
<=	denoting less than or equal to
<	denoting less than
¬<	denoting not less than

Bit-String Operators

The <u>bit-string operators</u> are:

¬	denoting not
&	denoting and
|	denoting or

String Operator

The <u>string operator</u> is:

||	denoting concatenation

Parentheses

<u>Parentheses</u> are used in expressions, for enclosing lists, and for specifying information associated with various keywords.

(	left parenthesis
)	right parenthesis

Separators and Other Delimiters

Name	Graphic	Use
comma	,	separates elements of a list
semicolon	;	terminates statements
assignment symbol	=	used in assignment statement and DO statement
colon	:	follows labels and condition prefixes; also used with dimension specifications
blank		used as a separator
quotation mark	'	encloses string constants and picture specifications

Name	Graphic	Use
period	.	separates items in qualified names; used as a decimal or binary point in constants
percent symbol	%	precedes compile-time statements
arrow	->	qualifies a reference to a based variable

DATA CHARACTER SET

Although the language character set is a fixed set defined for the language, the data character set has not been limited. Data may be represented by characters from the language set plus any other characters permitted by the particular machine configuration.

Any character that will result in a unique bit pattern is a valid character in the data character set, and may be used in source programs to construct character-string constants and comments.

COLLATING SEQUENCE

The collating sequence in PL/I is implementation-defined.

IDENTIFIERS

An identifier is a string of alphameric and break characters, not contained in a comment or constant, preceded and followed by a delimiter; the initial character must always be alphabetic.

Length of Identifiers

The maximum length of identifiers that a programmer constructs in writing a PL/I program is implementation defined.

KEYWORDS

A keyword is an identifier which is a part of the language. Keywords are not reserved words. They may be classified as follows:

statement identifiers

attributes

separating keywords

built-in function names

options

conditions

Some keywords may be written in an abbreviated form and these are listed in Appendix 4.

Statement Identifiers

A statement identifier is a sequence of one or more keywords used to define the function of a statement (see "Simple Statements").

Examples:

```
GO TO
DECLARE
READ
```

Attributes

Attributes are keywords that specify the characteristics of data, procedures, and other elements of the language.

Example:

```
FLOAT
RECURSIVE
SEQUENTIAL
```

Separating Keywords

The five separating keywords are used to separate parts of the IF and DO statements. They are THEN, ELSE, BY, TO, WHILE.

Built-in Function Names

A built-in function name is a keyword that is the name of an algorithm provided by the language and accessible to the programmer (see "Function References and Function Procedures" in Chapter 5).

Examples:

```
DATE
EXP
```

Options

An option is a specification that may be used by the programmer to influence the execution of a statement.

Examples:

```
TASK
BY NAME
```

Conditions

A condition is a keyword used in the ON, SIGNAL, and REVERT statements, and as a prefix to other statements (see "Prefixes"). The programmer may specify special action on occurrence of the condition (see "Interrupt Operations").

Examples:

```
OVERFLOW
ZERODIVIDE
```

THE USE OF BLANKS

Identifiers, constants (except character-string constants), picture specifications, composite operators (e.g., ┐=), and the class of dummy variables iSUB (see "The DEFINED Attribute" in Chapter 4) may not contain blanks.

Identifiers, constants, iSUB dummy variables, or picture specifications may not be immediately adjacent. They must be separated by a 60-character set operator, assignment symbol, percent symbol, arrow, parenthesis, colon, semicolon, comma, period, blank, or comment. Moreover, additional intervening blanks or comments are always permitted. Blanks are optional between keywords of the statement identifier GO TO.

Examples:

CALLA	is not equivalent to CALL A
A TO B BY C	is not equivalent to ATOBBYC
AB+BC	is equivalent to AB + BC

COMMENTS

General format:

```
/*[character-string]*/
```

Comments are normally used for documentation and do not participate in the execution of a program. A comment may be used wherever a blank is permitted (except in a character-string constant). The character string in a comment must not contain the character combination */ in that sequence.

Example:

```
LABEL:  /* THE BLOCK OF CODING BETWEEN
BEGIN-END IS USED FOR PAYROLL CALCULA-
TIONS */
          BEGIN;
          .
          .
          .
          END;
```

BASIC PROGRAM STRUCTURE

A PL/I program is constructed from basic program elements called statements.

Statements are grouped into larger program-elements, the group and the block. There are two types of statements: simple and compound.

SIMPLE STATEMENTS

A simple statement is defined as:

```
[[statement-identifier]
statement-body] ;
```

The "statement identifier," if it appears, is a keyword , characterizing the kind of statement. If it does not appear, and the statement body does appear, then the statement is an assignment statement. If only the semicolon appears, the statement is called a null statement.

Examples:

DO I = J TO 10;	(DO is the keyword)
A = B + C;	(assignment statement)
;	(null statement)

COMPOUND STATEMENTS

A compound statement is a statement that contains other program-elements. There are two of them:

The IF compound statement

The ON compound statement

The final contained statement of a compound statement is a simple statement and thus has a terminal semicolon. Hence, the compound statement will automatically be terminated by this semicolon.

Examples:

```
IF A=B THEN GO TO S1; ELSE A=C;

ON OVERFLOW GO TO OVFIX;
```

Each PL/I statement is described in the alphabetic list of statements in Chapter 8.

PREFIXES

There are two types of prefixes: label prefixes and condition prefixes.

Label Prefixes

Statements may be labeled to permit reference to them. A labeled statement has the following form:

```
identifier:[identifier:]...statement
```

The one or more "identifiers" are called labels and may be used interchangeably to refer to that statement.

Labels appearing before PROCEDURE and ENTRY statements are special cases and are known as entry names (see "Procedure References"). All other labels are called statement labels.

A label appearing before a statement is said to be declared, by virtue of its appearance as a label.

Statement labels appearing before DECLARE statements are ignored.

Condition Prefixes

A condition prefix specifies whether or not a program interrupt will result upon the occurrence of the specified condition. (For information regarding the use of the condition prefix see the section "Interrupt Operations" in Chapter 6.)

One or more condition prefixes may be attached to a statement.

Each condition prefix is followed by a colon to separate it from the rest of the statement or from other prefixes; condition prefixes precede the entire statement, including any possible label prefixes for the statement.

A condition prefix is a list of condition names, separated by commas and enclosed in parentheses. Thus, a statement with a set of prefixes has the following general form:

```
{(condition-name [,condition-
   name]...):}...[label:]...
   statement
```

The condition names are chosen from the following fixed set:

```
UNDERFLOW
OVERFLOW
ZERODIVIDE
FIXEDOVERFLOW
CONVERSION
SIZE
STRINGRANGE
SUBSCRIPTRANGE
CHECK (identifier-list)
```

Note: CHECK (identifier list) may be used as a prefix only with the PROCEDURE and BEGIN statements.

The meanings of these conditions are explained in "The ON Statement," in Chapter 8.

Any of these condition names may be preceded by the word NO. If NO is used, there can be no intervening blank between NO and the condition. For example, NOCONVERSION can be specified in the prefix list.

GROUPS

A group is a collection of one or more statements and is used for control purposes.

A group has one of two forms. The first form, called a DO-group, is:

```
[label:] . . . DO-statement
               program-element-1
               program-element-2
                     .
                     .
                     .
               END [label];
```

The label following END is one of the labels of the DO statement (see "Use of the END Statement" in this chapter).

The DO statement is called the heading statement of the DO-group, and may specify iteration. Each program element represents one or more statements.

The second form of a group is simply a single statement, as follows:

```
[label:] . . . statement
```

The "statement" is any statement except DO, END, PROCEDURE, BEGIN, DECLARE, FORMAT, or ENTRY.

Example of the first form:

```
ALPHA: DO;
       A=B*C;

       IF A < 0 THEN DO; B=1; C=0; END;

       END ALPHA;
```

In the example above, any of the single statements -- except the DO and END statements -- is an example of the second form of a group.

BLOCKS

A block is a collection of statements that defines the program region -- or scope -- throughout which an identifier is established as a name. It also is used for control purposes.

There are two kinds of blocks, begin blocks and procedure blocks.

A begin block has the general form:

```
[label:] . . . BEGIN-statement
               program-element-1
               program-element-2
                     .
                     .
                     .
               END [label];
```

The label following END is one of the labels of the BEGIN statement (see "Use of the END Statement" in this chapter).

A procedure block, or procedure, has the general form:

```
label:  [label:] . . . PROCEDURE-statement
                       program-element-1
                       program-element-2
                             .
                             .
                             .
                       END [label];
```

The label following END is one of the labels of the PROCEDURE statement (see "Use of the END Statement" in this chapter).

The BEGIN statement and the PROCEDURE statement in the above forms are called heading statements.

While the labels of the BEGIN statement are optional, the PROCEDURE statement must have at least one label.

Although the begin block and the procedure have a physical resemblance and play the same role in delimiting scope of names (see "Scope of Declarations," in Chapter 4) and defining allocation and freeing of storage (see "Allocation of Data and Storage Classes," in Chapter 6), they differ in an important functional sense. A begin block, like a single statement, is activated by normal sequential flow (except when used as an on-unit), and it can appear wherever a single statement can appear. A procedure can only be activated remotely by CALL statements, by statements in which a CALL option appears, or by function references. When a program containing a procedure is executed, control passes around the procedure, from the statement before the PROCEDURE statement to the statement after the END statement of the procedure.

Since a procedure can be activated only by a reference to it, every procedure must have a name. The label required for the heading statement of a procedure serves as the procedure name. More than one label provides more than one procedure name.

The procedure name gives a means of activating the procedure at its primary entry point. Secondary entry points can also be defined for a procedure by use of the ENTRY statement. The labels preceding all ENTRY statements in a given procedure and the heading statement of the procedure are collectively called entry names for the procedure.

As the above definition of block implies, any block A can include another block B, but partial overlap is not possible; block B must be completely included in block A. Such nesting may be specified to any depth.

A procedure that is not included in any other block is called an external procedure. A procedure included in some other block is called an internal procedure.

Every begin block must be included in some other block. Hence, the only external blocks are external procedures.

All of the text of a begin block except the labels preceding the heading statement of the block is said to be contained in the block.

All of the text of a procedure except the entry names of the procedure is said to be contained in the procedure.

That part of the text of a block B that is contained in block B, but not contained in any other block contained in B, is said to be internal to block B.

The entry names of an external procedure are not internal to any procedure and are called external names.

The notion of internal to is vital in the definition of scope (see "Scope of Declarations" in Chapter 4).

Example:

```
 A:  PROCEDURE;                      ┐
     statement 1                     │
     B:  BEGIN;                      │
         statement 2┐                │
         statement 3┘                │
         END B;                      │
     statement 4                     │
     C:  PROCEDURE;               ┐  │
         statement 5              │  │
     X:  ENTRY;                   │  │
         D:  BEGIN;               │  │
             statement 6┐         │  │
             statement 7┘         │  │
             END D;               │  │
         statement 8              │  │
         END C;                   ┘  │
     statement 9                     ┘
     END A;
```

In this example, statements 1 through 9 are labeled or unlabeled simple statements.

As the brackets on the right indicate, block A contains block B and block C, and block C contains block D.

Block A is an external procedure. The procedure name is A, which is an external name, and the only entry name for the procedure.

X is an entry name corresponding to a secondary entry point for procedure C.

Blocks B and D are begin blocks.

Block C is an internal procedure.

The text internal to block A consists of

```
PROCEDURE;
statement 1
B:
statement 4
C:
X:
statement 9
END A;
```

The text internal to block B consists of

```
BEGIN;
statement 2
statement 3
END B;
```

The text internal to block C consists of

```
PROCEDURE;
statement 5
ENTRY;
D:
statement 8
END C;
```

The text internal to block D consists of

```
BEGIN;
statement 6
statement 7
END D;
```

USE OF THE END STATEMENT

As the examples above imply, the END statement has the form:

```
END [label];
```

and is used to terminate a group or a block.

If the optional label following END is not used, the END statement terminates that unterminated group or block headed by the DO, BEGIN, or PROCEDURE statement that physically precedes, and appears closest to, the END statement.

If, however, a label (e.g., L) is used following END, the statement terminates that unclosed group or block headed by the DO, BEGIN, or PROCEDURE statement with the label L that physically precedes, and appears closest to, the END statement. Any groups or blocks headed by DO, BEGIN, or PROCEDURE statements contained in the terminated block L are also automatically terminated by the END statement END L. This feature eliminates the necessity of writing the intermediate END statements to terminate the contained blocks and groups.

The statement labeled L, which heads the group or block terminated by the END statement END L, is internal to a certain block in the program (see "Blocks," for a definition of <u>internal to</u>). The terminating statement END L, together with its own possible statement-labels, is also considered to be internal to the same block. (If the statement labeled L is a BEGIN or PROCEDURE statement, this block is, of course, the block L.)

The END statement may itself be labeled, and a reference to this label can be made from any part of the program where the label is known. (For a definition of <u>known</u>, see "Basic Rule on Use of Names" in Chapter 4).

Example:

```
(a)   A:  PROCEDURE;
               .
               .
               .
      B:  BEGIN;
               .
               .
               .
      A:  PROCEDURE;
               .
               .
               .
      C:  DO;
               .
               .
               .
      X:  END B;
      END A;
```

```
(b)   A:  PROCEDURE;
               .
               .
               .
      B:  BEGIN;
               .
               .
               .
      A:  PROCEDURE;
               .
               .
               .
      C:  DO;
               .
               .
               .
      END;
      END;
      X:  END;
      END;
```

In example (a), the statement X:END B terminates the DO group, the internal procedure A, and the block B. The statement END A terminates the external procedure A. The statement X:END B is internal to block B.

Example (b) is equivalent to example (a).

PROGRAMS

A <u>program</u> is composed of one or more external procedures.

Information that is operated on in a PL/I object program during execution is called data. Each data item has a definite type and representation.

The aim of this chapter is to present a discussion of (1) the various organizations that data may have, (2) the methods by which data can be referred to, and (3) the types of data allowed.

DATA ORGANIZATION

Data may be organized as scalar items (i.e., single data items) or aggregates of data items (i.e., arrays and structures). File names, entry names, and programmer-defined condition names are not considered to be data.

SCALAR ITEMS

A data item may be either a constant or the value of a scalar variable. Constants and scalar variables are called scalar items. Scalar variables and scalar data items may also be called element variables and element data items respectively.

Constants

A constant is a data item that denotes itself, i.e., its representation is both its name and its value; thus, it cannot change during the execution of a program. Each constant has a type, as described later in this chapter. A signed constant is an arithmetic constant preceded by one of the prefix operators + or -. Wherever the word "constant" appears alone, and refers to an arithmetic constant, it is to be assumed to refer to an unsigned constant.

Scalar Variables

A scalar variable, like a constant, denotes a data item. This data item is called the value of the scalar variable. Unlike a constant, however, a variable may take on more than one value during the execution of a program. The set of values that a variable may take on is the range of the variable. The range of a variable is always restricted to one data type (and, if the type is arithmetic, to one base, scale, mode, and precision - see "Arithmetic Data" in this chapter). If there are no further restrictions declared for the range, the variable may assume values over the entire set of data of that type.

Reference is made to a scalar variable by a name, which may be a simple name, a subscripted name, a qualified name, or a subscripted qualified name (see "Naming" in this chapter).

DATA AGGREGATES

In PL/I, all classes of variable data items may be grouped into arrays or structures. Rules for this grouping are given below. (For the method of referring to an array or structure or a particular item of an array or structure, see "Naming" in this chapter.)

Arrays

An array is an n-dimensional, ordered collection of elements, all of which have identical data declarations. (If arithmetic, all of the elements of the array must have the same base, scale, mode, and precision or the same picture. If character-string or bit-string, all of the elements must have the same actual length, if fixed length, or the same maximum length, if varying length.) The number of dimensions of an array, and the upper and lower bounds of each dimension, are specified by the use of the dimension attribute.

Example:

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

This statement defines A as an array with 2 dimensions: 3 rows and 4 columns. The matrix given below illustrates the array A.

```
A(1,1)   A(1,2)   A(1,3)  A(1,4)
A(2,1)   A(2,2)   A(2,3)  A(2,4)
A(3,1)   A(3,2)   A(3,3)  A(3,4)
```

The elements of an array may be structures (see "Arrays of Structures").

Structures

A structure is a hierarchical collection of scalar variables, arrays, and structures. These need not be of the same data type nor have the same attributes.

The outermost structure is a major structure, and all contained structures are minor structures.

A structure is specified by declaring the major structure name and following it with the names of all contained elements. Each name is preceded by a level number, which is a non-zero decimal integer constant. A major structure is always at level one and all elements contained in a structure (at level *n*) have a level number that is numerically greater than *n*, but they need not necessarily be at level n+1, nor need they all have the same level number.

A minor structure at level *n* contains all following items declared with level numbers greater than *n* up to but not including the next item with a level number less than or equal to *n*. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a DECLARE statement.

Examples:

1. DECLARE 1 PAYROLL, 2 NAME, 2 HOURS, 3 REGULAR, 3 OVERTIME, 2 RATE;

   takes the form:

```
1 PAYROLL
   2 NAME
   2 HOURS
      3 REGULAR
      3 OVERTIME
   2 RATE
```

In the above example PAYROLL is defined as the major structure containing the scalar variables NAME and RATE and the structure HOURS. The structure HOURS contains the scalar variables REGULAR and OVERTIME.

2. DECLARE 1 A, 2 B, 2 C, 3 D (2), 3 E, 2 F;

   This takes the form:

```
A
    B
    C
        D(1)
        D(2)
        E
    F
```

The decimal integers before the identifiers specify the levels; the decimal integer in parentheses specifies the bounds of the one-dimensional array. A is defined as the major structure and contains the minor structure C and the scalar variables B and F. C contains D, a one-dimensional array with two scalar variables, and the scalar variable E.

3. DECLARE 1 A, 3 B, 2 C;

   This takes the form:

```
A
    B
    C
```

   Note that B and C are at the same level although their level numbers differ.

Arrays of Structures

An array of structures is formed by giving the dimension attribute to a structure.

Examples:

1. DECLARE 1 CARDIN(3), 2 NAME, 2 WAGES, 3 NORMAL, 3 OVERTIME;

   The decimal integers before the identifiers specify the level. The name, CARDIN, represents an array of structures. Because CARDIN has a dimension specified, NAME, NORMAL, and OVERTIME are arrays, and their elements are referred to by subscripted names.

   The form of the data is as follows:

```
CARDIN  (1)  NAME   (1)
             WAGES  (1)  NORMAL    (1)
                         OVERTIME  (1)

CARDIN  (2)  NAME   (2)
             WAGES  (2)  NORMAL    (2)
                         OVERTIME  (2)

CARDIN  (3)  NAME   (3)
             WAGES  (3)  NORMAL    (3)
                         OVERTIME  (3)
```

2. DECLARE 1 X, 2 Y, 2 Z (2), 3 P (2,2), 3 Q, 2 R;

X is an undimensioned major structure containing scalar variables, arrays, and a structure.
Y is a scalar variable
Z is an array of structures
P is a three-dimensional array
Q is a one-dimensional array
R is a scalar variable

The form of the data is as follows:

```
     ┌ Y
     │         ┌P (1,1,1)
     │         │P (1,1,2)
     │ Z (1)   │P (1,2,1)
     │         │P (1,2,2)
     │         └Q (1)
X    │
     │         ┌P (2,1,1)
     │         │P (2,1,2)
     │ Z (2)   │P (2,2,1)
     │         │P (2,2,2)
     │         └Q (2)
     │ R
     └
```

Attributes of Structures

Structures and arrays of structures are not given data attributes. These can be given only to structure base elements.

Major structure names may be declared with the EXTERNAL attribute. Items contained in structures may not be declared with the EXTERNAL attribute, and even if INTERNAL is unspecified, they are assumed to be INTERNAL.

All items in the same structure are of the same storage class, since only the major structure may be given a storage-class attribute. The storage class of the major structure applies to all elements of the structure. If a structure has either the CONTROLLED or the BASED attribute, only the major structure, not its elements, may be allocated and freed.

NAMING

This section describes the rules for referring to a particular data item, groups of items, arrays, and structures. The permitted types of data names are simple, qualified, subscripted, and subscripted qualified.

SIMPLE NAMES

A simple name is an identifier (see "Identifiers," in Chapter 1) that refers to a scalar, an array, or a structure.

SUBSCRIPTED NAMES

A subscripted name is used to refer to an element or a cross section of an array. It is a simple name that has been declared to be the name of an array followed by a list of subscripts. The subscripts are separated by commas and are enclosed in parentheses. A subscript is an asterisk or a scalar expression that is evaluated and converted to an integer before use (see "Evaluation of Expressions," in Chapter 3). The number of subscripts must be equal to the number of dimensions of the array, and the value of a specified subscript must fall within the bounds declared for that dimension of the array.

A subscripted name takes the form:

identifier (subscript [, subscript] ...)

Examples:

```
A (3)
FIELD (B,C)
PRODUCT (SCOPE * UNIT + VALUE, PERIOD)
ALPHA (1,2,3,4)
X(1,*,3)
```

Cross Sections of Arrays

The concept of cross sections is a logical extension of the subscripting notation. A cross section of an array is referred to by the array name, followed by a list of subscripts, at least one of which is an asterisk. The subscripts are separated by commas, and the entire list is enclosed in parentheses. The number of items in the list must be equal to the number of dimensions of the array. If the array is of dimensionality n, then an asterisk may appear in k ≤ n positions. If the jth list position is occupied by an asterisk, the cross section of the array includes elements covered by varying the jth subscript between its bounds. The dimensionality of the cross section is equal to the number of asterisks, k, in the subscript list. If all subscript positions are occupied by asterisks, then this reference to the cross section is equivalent to a reference to the entire array.

A cross section may be used anywhere that the name of an array of dimensionality k is required. Subsequent references to the word "array" in this document should therefore be taken to include cross sections of arrays.

Examples:

1. A (3,*) denotes the third row of the array A.

2. B (*, *, 2) is a two-dimensional cross section and denotes the second plane of the array B.

3. If MATRIX is the array:
```
1  2  3
4  5  6
7  8  9
```
   MATRIX (*, 2) represents the array:
```
   2
   5
   8
```

QUALIFIED NAMES

A simple name usually refers uniquely to a scalar variable, an array, or a structure. However, it is possible for a name to refer to more than one variable, array, or structure if the identically named items are themselves parts of different structures. In order to avoid any ambiguity in referring to these similarly named items, it is necessary to create a unique name; this is done by forming a <u>qualified name</u>. This means that the name common to more than one item is preceded by the name of the structure in which it is contained. This, in turn, can be preceded by the name of its containing structure, and so on, until the qualified name refers uniquely to the required item. The section "Multiple Declarations and Ambiguous References" in Chapter 4, contains further information on this subject.

Thus, the qualified name is a sequence of names specified left to right in order of increasing level numbers; the names are separated by periods, and blanks may be placed as desired around the periods. The sequence of names need not include all of the containing structures, but it must include sufficient names to resolve any ambiguity. Any of the names may be subscripted.

The qualified name, once composed, is itself a name. Subsequently, in this publication, when the terms scalar variable name, array name, or structure name are used they should also be taken to include qualified names.

A qualified name takes the form:

```
identifier {. identifier} ...
```

Examples:

1. A program may contain the structures:
```
DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRIP-
  TION, 2 PRICE;
DECLARE 1 CARDOUT, 2 PARTNO, 2 DES-
  CRIPTION, 2 PRICE;
```
   Elements are then referred to as:
```
      CARDIN.PARTNO
      CARDOUT.PARTNO
      CARDIN.PRICE
```

2. A program may contain the structure:
```
DECLARE 1 MARRIAGE, 2 MAN, 3 NAME, 3
  DATE, 2 WOMAN, 3 NAME, 3 DATE;
```
   Elements are then referred to as:
```
      MAN.NAME
        or MARRIAGE.MAN.NAME

      WOMAN.NAME
        or MARRIAGE.WOMAN.NAME
```

3. If the same program also contains the structure:
```
      DECLARE 1 BIRTH, 2 WOMAN, 3 NAME,
        3 DATE, 2 ADDRESS;
```
   Elements are then referred to as:
```
      MAN.NAME
        or MARRIAGE.MAN.NAME

      MARRIAGE.WOMAN.NAME

      BIRTH.NAME
        or BIRTH.WOMAN.NAME

      ADDRESS
```
   and the minor structures referred to as:
```
      MARRIAGE . WOMAN

      BIRTH . WOMAN
```

SUBSCRIPTED QUALIFIED NAMES

The elements of an array contained in a structure and requiring name qualification for identification are referred to by <u>subscripted qualified names</u>. A subscripted qualified name is a sequence of names and subscripted names separated by periods.

The order of names is as given for any qualified name. The subscript list following each name refers to the dimensions associated with the name if the name is declared to be the name of an array in the structure description.

As long as the order of the subscripts remains unchanged, subscripts may be moved to the right or left and attached to names at a lower or higher level, respectively. The number of subscripts, if any are specified, must match the number of dimensions of the array. A subscripted qualified name takes the general form:

```
     identifier [ (subscript [, subscript]
          ...)]
        {. identifier [(subscript [, sub-
        script]...)] }...
```

If any subscripts are given in a reference to a qualified name, all those subscripts which apply to dimensions of containing structures must be given.

Example 1:

A is an array of structures with the following description:

```
     DECLARE 1 A (10,12), 2 B (5), 3 C (7),
        3 D;
```

The following subscripted qualified names refer to the same element, which is the seventh element of C contained in the fifth element of B contained in tenth row and twelfth column of A:

```
     (1)  A (10,12) .  B (5) .  C (7)
     (2)  A (10) .  B (12,5) .  C (7)
     (3)  A (10) .  B (12) .  C (5,7)
     (4)  A .  B (10,12,5) .  C (7)
     (5)  A .  B (10,12) .  C (5,7)
     (6)  A .  B (10) .  C (12,5,7)
     (7)  A .  B .  C (10,12,5,7)
     (8)  A (10,12) .  B .  C (5,7)
     (9)  A (10) .  B .  C (12,5,7)
     (10) A (10,12,5,7) .  B .  C
```

If structure B, but not structure A, is necessary for unique identification of this use of C, any of forms (4), (5), (6), or (7) may be used without including the A.

If structure A, but not B, is necessary for identification of C, forms (7), (8), (9), or (10) may be used without including the B.

Example 2:

If FIELD is the array of structures:

```
      DECLARE 1 FIELD(3),
                  2 STATUS,
                  2 VALUE;
```

then FIELD(*).STATUS represents the array:

```
     FIELD(1).STATUS
     FIELD(2).STATUS
     FIELD(3).STATUS
```

DATA TYPES

The types of data allowed by PL/I can be categorized as problem data and program-control data.

PROBLEM DATA

Problem data is any data that can be classified as type arithmetic or type string.

Arithmetic Data

An arithmetic data item is one that has a numeric value with characteristics of base, scale, mode, and precision. The data item may be represented either as a numeric field or in a coded form, that is, in an internal representation that is implementation dependent. A numeric field is a string of characters or bits that is given a numeric interpretation by means of the PICTURE attribute (see Chapter 4). The base, scale, and precision are all specified in the picture of the numeric field. A data item in coded form does not have a PICTURE attribute, but has its characteristics given by the attributes specifying base, scale, mode, and precision. An arithmetic constant is of coded form.

Base (decimal or binary), scale (fixed-point or floating-point), and precision have reference to internal representation of the data described and to the internal arithmetic that is to be used.

BASE: Arithmetic data may be specified as having either decimal or binary base.

SCALE: Arithmetic data may be specified as having either fixed-point or floating-point scale. Fixed-point data items are rational numbers for which the number of decimal or binary digits is specified; the position of the decimal or binary point may also be specified by a scale factor. Floating-point data items are rational numbers in the form of a fractional part and an exponent part.

MODE: Arithmetic data may be operated on in either the real or complex mode. In the complex mode, a data item is considered to consist of a number pair, the first member of the pair representing the real part of the complex number and the second, the imaginary part.

PRECISION: The precision of fixed-point data (p,q) is specified by giving the total number of binary or decimal digits, p, to be maintained and a scale factor, q. The precision of floating-point data is specified by giving the effective number, p, of binary or decimal digits to be maintained in the fractional part (for an implementation, the actual number of digits maintained internally may be greater than p). Note that p must be greater than zero.

Real Arithmetic Constants

A real arithmetic constant is either binary or decimal.

DECIMAL FIXED-POINT CONSTANTS: A decimal fixed-point constant is represented by one or more decimal digits with an optional decimal point. If a decimal point is not specified, the constant is a decimal integer constant.

Examples:

```
72.192
.308
255.
158
```

BINARY FIXED-POINT CONSTANTS: A binary fixed-point constant is represented by one or more binary digits with an optional binary point followed by the letter B.

Examples:

```
10.B
11011B
11.1101B
.001B
```

STERLING FIXED-POINT CONSTANTS: Sterling quantities may be specified and will be interpreted as decimal fixed-point pence. A sterling fixed-point constant consists of the following concatenated fields:

- a pounds field that is a decimal integer
- a decimal point
- a shillings field is one or more decimal digits that represent a decimal integer less than 20
- a decimal point
- a pence field that is one or more decimal digits with an optional decimal point (the integral part must be less than 12.)
- an L

Examples:

```
101.13.8L
1.10.0L
0.0.2.5L
```

DECIMAL FLOATING-POINT CONSTANTS: A decimal floating-point constant is represented by one or more decimal digits with an optional decimal point, followed by the letter E, followed by an optionally signed exponent. The exponent is one or more decimal digits specifying an integral power of ten.

Examples:

```
12.E23
317.5E-16
0.1E+3
.42E+73
32E-5
```

BINARY FLOATING-POINT CONSTANTS: A binary floating-point constant is represented by one or more binary digits with an optional binary point, followed by the letter E, followed by an optionally signed exponent, followed by the letter B. The exponent is one or more decimal digits specifying an integral power of two.

Examples:

```
1.1011E3B
.11011E-27B
```

PRECISION OF REAL ARITHMETIC CONSTANTS: For purposes of expression evaluation, an apparent precision is defined for real arithmetic constants.

Real fixed-point constants have a precision (p,q) where p is the total number of digits in the constant and q is the number of digits specified to the right of the decimal point.

The precision of a sterling constant is equivalent to the precision of its corresponding value in fixed-point pence. This value is determined as follows: multiply the value of the pounds field by 240; add the product of 12 and the value of the shillings field; add the value of the pence field. The precision of the result (with leading zeros removed) is the precision of the corresponding sterling constant.

The precision of a floating-point constant is (p) where p is the number of digits of the constant left of the E.

Examples:

```
3.14 has precision (3,2)
0.012E5 has precision (4)
0.9.0.5L has precision (4,1)
0000001B has precision (7,0)
```

Imaginary Arithmetic Constants

An imaginary constant represents a complex value of which the real part is zero and the imaginary part is the value specified.

It is represented by a real arithmetic constant, other than a sterling constant, followed by the letter I. PL/I does not define complex constants with non-zero real parts, but provides the facility to specify such data through an expression, e.g., 10.1+9.2I.

Examples:

```
27I
3.968E10I
```

Arithmetic Variables

Arithmetic variables are names of arithmetic data items. These names have been given the characteristics (i.e., attributes) of base, scale, mode, and precision (see Chapter 4).

String Data

String data can be classified as character-string or bit-string. The length of a string data item is equivalent to the number of characters (for a character-string) or the number of binary digits (for a bit-string) in the item. A string data item of length zero is known as a null string.

Character-String Data

Character-string data consists of a string of zero or more characters in the data character set (see "Data Character Set," in Chapter 1). The string may be fixed or varying in length. The actual number of characters must be specified if it is of fixed length, and the maximum length must be specified if it is of varying length.

CHARACTER-STRING CONSTANTS: A character-string constant is zero or more characters in the data character set enclosed in quotation marks. If it is desired to represent a quotation mark, it must appear as two immediately adjacent quotation marks. The constant may optionally be preceded by a decimal-integer constant in parentheses to specify repetition. If the constant specifying repetition is zero, the result is the null character string.

In a string repetition factor, blanks may optionally surround the decimal integer constant, or they may separate the right parenthesis and leading quote.

A character string constant may contain a string of characters which syntactically constitute a comment; however, these characters are treated as part of the string value rather than as a comment.

Examples:

```
'$ 123.45'
'JOHN JONES'
'IT''S'
(3)'TOM'
''
```

The fourth is exactly equivalent to

```
'TOMTOMTOM'
```

The last example, which is two single quotation marks with no intervening blank, specifies the null character string.

Bit-String Data

Bit-string data consists of a string of zero or more binary digits (0 and 1). The string may be fixed or varying in length. The actual length of the field must be specified if it is of fixed length, and the maximum length must be specified if it is of varying length.

BIT-STRING CONSTANTS: A bit-string constant is zero or more binary digits enclosed in quotation marks, followed by the letter B. The constant may optionally be preceded by a decimal-integer constant in parentheses, to specify repetition. If the constant specifying repetition is zero, the result is the null bit string.

Examples:

```
'0100'B
(10)'1'B
''B
```

The second is exactly equivalent to

```
'1111111111'B
```

The last example specifies the null bit string.

String Variables

String variables are names of string data items. These names have been given string attributes.

PROGRAM-CONTROL DATA

Program-control data is any data that can be classified as type label, task, event, pointer, area, or cell.

Label Data

Statement-label data is used only in connection with statement labels. Statement label data may be constants or variables, and the variables may be elements of structures or arrays.

Statement-Label Constants

A statement-label constant is an identifier that appears in the program as a statement label. It permits references to be made to statements.

Example:

```
                    .
                    .
                    .
 ROUTINE1:   IF X > 5 THEN GO TO EXIT;
                    .
                    .
                    .
             GO TO ROUTINE1;
                    .
                    .
                    .
             EXIT: RETURN;
```

ROUTINE1 and EXIT are statement-label constants.

Statement-Label Variables

A <u>statement-label variable</u> is a variable that has as values statement-label constants. These variables can be grouped into arrays or structures.

Example:

```
          DECLARE X LABEL;
          X = POSROUTINE;
                  .
POSROUTINE:       .
                  .
          X = NEGROUTINE;
          GO TO X;
                  .
NEGROUTINE:       .
                  .
```

The label variable X may have the value of either POSROUTINE or NEGROUTINE, both labels in the procedure. In the above example, GO TO X transfers control to NEGROUTINE.

A statement-label constant or a scalar label variable is called a <u>statement-label designator</u>.

Task Data

A task variable is the name of a task (see "Asynchronous Operations and Tasks" in Chapter 6, and "The TASK Attribute" in Chapter 4). A task variable may be an element of an array or of a structure. The priority associated with a task variable may be assigned in the CALL statement, or in an assignment statement via the PRIORITY pseudo-variable (see Chapter 8).

Event Data

An event variable is the name of an event used in connection with asynchronous processing, in multitasking, the DISPLAY statement, or with record-oriented I/O operations. An event variable may be an element of an array or of a structure.

An event variable has associated <u>completion</u> and <u>status</u> values that can be accessed by the COMPLETION and STATUS built-in functions (see "The EVENT Attribute" in Chapter 4).

Locator Data

Locator data consists of pointer variables and offset variables. A pointer variable has a value that is used to identify the location of a single generation of a variable. An offset variable has a value that is used to identify the location of a based variable relative to the beginning of an area. (See "OFFSET and POINTER" in Chapter 4.)

Locator Qualification

Locator qualification is used to associate one or more pointer or offset values with a based variable to identify a particular generation of data. If a based variable is referred to without a locator qualifier, the reference is the same as a reference qualified by the locator variable declared with the based variable in the BASED attribute specification.

The format of a locator qualifier is as follows:

```
  scalar-locator-expression ->
    [based-locator-variable ->]...
     based-variable
```

where "scalar-locator-expression" is a pointer-variable, an offset-variable, or a function reference that returns a pointer or offset value.

General rules:

1. Locator qualification is used to indicate the generation of a based variable to which the associated reference applies.

2. If an offset expression or an offset variable is used as a locator qualifier, its value is implicitly converted to a pointer value.

3. More than one locator qualifier can be specified in a reference. Only the first (or leftmost) can be a function reference; all other locator qualifiers must themselves be based variables.

4. If more than one qualifier is used, they are read from left to right.

Examples:

```
A = P->B;
A = P->Q->B;
A = ADDR(X)->B;
```

The first example causes assignment to A of the value of B in the generation pointed to by P. The second example specifies that the value of P is to be used to locate the generation of Q which locates the specific generation of B to be assigned to A. In the third example, the generation of B is derived from the location of the variable X.

Area Data

An area variable represents an area of storage in which based variables may be allocated and freed.

Cell Data

A cell is a unit of storage that may be used to hold values of different data types. However, only the value of the most recently assigned data type can be accessed.

Cells are organized in the same way that structures are organized; the name of the cell must be at a higher level than its alternatives. For example, the following statement specifies that the storage allocated for the cell named ALPHA may contain either of the two alternatives, ALT1 (a bit string) or ALT2 (a structure), but not both at the same time.

```
DECLARE 1 ALPHA CELL,
          2 ALT1 BIT (60),
          2 ALT2,
             3 BETA FLOAT,
             3 GAMMA FIXED;
```

A cell provides storage equivalence and not data equivalence. In other words, since only one alternative can be active at one time, the value of that alternative cannot be retrieved by a reference to another alternative. The assignment of a value to an alternative deactivates the previously active alternative and in effect strips it of its value.

Thus, the value of an alternative can only be retrieved by a reference to that alternative. The cell name may be used to qualify the reference but a reference to the cell name alone will retrieve no value.

EXPRESSIONS

An expression is an algorithm used for computing a value. Expressions are of the three types: scalar, array, and structure, depending upon the type of the result. An array (or structure) expression is simply an array (or structure) evaluated by expansion of the expression into a collection of scalar expressions. Syntactically, a scalar expression consists of a constant, a scalar variable, a scalar expression enclosed in parentheses, a scalar expression preceded by a prefix operator, two scalar expressions connected by an infix operator, or a function reference that returns a scalar value. (Note that any programmer-written function returns a scalar value, but some built-in functions may return array or structure values.) Operands in a scalar expression need not have the same data attributes. If they differ, conversion will be performed before the operation.

SCALAR EXPRESSIONS

A scalar expression returns a scalar value. The class of the expression is dependent upon the operators -- arithmetic, comparison, bit string, and concatenation. In the case of program control data, the operands determine the class of expression. Only the operators = and ┐= may appear with pointer and offset. No operators may appear with label, cell, area, event, and task data.

If A and B are expressions, then the operators + and - used in expressions of the form +A or -A, are called prefix operators. When these operators are used in expressions of the form A+B or A-B they are called infix operators.

Arithmetic Operations

An arithmetic expression of any complexity is composed of a set of elementary arithmetic operations.

An elementary arithmetic operation has the following general format:

```
{{+|-} operand} | {operand
   {+| - | * | / | **} operand}
```

The general format specifies the prefix operations of plus and minus and the infix operations of addition, subtraction, multiplication, division, and exponentiation. Operations are performed only with coded arithmetic data. If necessary, the data will be converted to coded arithmetic type before the operation is performed.

Mixed Characteristics

The two operands of an arithmetic operation may differ in form, base, scale, mode, and precision. When they differ (except in some cases of exponentiation), conversion takes place according to the following rules:

FORM: Numeric field operands of arithmetic operations will be converted to coded form. The result of an arithmetic operation is always in coded form.

BASE: If bases differ, the decimal operand is converted to binary.

SCALE: If the scales of the operands differ, the fixed-point operand will be converted to floating-point, except in the case of exponentiation in which the first operand is floating-point and the second is fixed-point with precision (p,0). In the latter case, the second operand is not converted, and the result has the base, scale, mode, and precision of the first operand.

MODE: If the modes differ, the real operand is converted to complex mode (by acquiring an imaginary part of zero with the same base, scale, and precision as the real part). However, when the operation is exponentiation and the second operand is fixed-point with precision (p,0), then the second operand is not converted.

PRECISION: If precisions differ, no conversion is done.

Results of Arithmetic Operations

After the conversions specified above have taken place, the arithmetic operation is performed. Any necessary truncations will be made towards zero, regardless of the base or scale of the operands.

The base, scale, mode, and precision of the result depend on the operands and the operator in the following ways:

1. Prefix operations: The prefix operations of plus and minus yield a result having the base, scale, mode, and precision of the operand.

2. Floating-point: If the operands of an infix operation are floating-point the result is floating-point, and the base and mode of the result are the common base and mode of the operands. The precision of the result is the greater of the precisions of the two operands.

3. Fixed-point: If the operands of an infix operation are fixed, and if the operation is not exponentiation, the result is fixed, and the base and mode of the result are the common base and mode of the operands. If the operation is exponentiation, the second operand is converted to floating point if its scale factor is not zero; and the first operand is converted to floating-point unless the second operand is an unsigned integer constant meeting the conditions of item d below; in these cases, the rules for floating-point apply.

   The precision of a fixed-point result depends on the operation and the precisions of the operands, according to rules given below. The following symbols are used:

N	the maximum precision allowed by the implementation for the base of the result
m	the total number of positions in the result
n	the scale factor of the result
p	the total number of positions in operand one
q	the scale factor of operand one
r	the total number of positions in operand two
s	the scale factor of operand two
y	value of operand two, if it is an unsigned integer constant

   a. Addition and subtraction:

      m = min(N,max(p-q,r-s)+max(q,s)+1)
      n = max(q,s)

   b. Multiplication:

      m = min(N,p+r+1)
      n = q+s

   c. Division:

      m = N
      n = N-p+q-s

   d. Exponentiation: if the second operand is an unsigned non-zero real fixed-point constant of precision (r,0),

      m = (p+1) *y - 1
      n = q *y

      If m>N, however, or y is not an unsigned non-zero real fixed-point constant of precision (r,0), the first operand is converted to floating-point and rules for floating-point exponentiation apply.

   e. The above rules hold for both real and complex mode.

Note: Some special cases of exponentiation are defined as follows:

1. Real Mode, x_1**x_2:

   a. If $x_1=0$ and $x_2>0$, the result is 0.

   b. If $x_1=0$ and $x_2\leq0$, the ERROR condition is raised.

   c. If $x_1\neq0$ and $x_2=0$, the result is 1.

   d. If $x_1<0$ and x_2 is not fixed-point with precision (p,0), the ERROR condition is raised.

2. Complex Mode, z_1**z_2

   a. If $z_1=0$ and z_2 has its real part >0 and its imaginary part equal to 0, the result is 0.

   b. If $z_1=0$ and the real part of z_2 is not greater than 0 or the imaginary part of z_2 is not equal to 0, the FRROR condition is raised.

Arithmetic Conversions

1. Arithmetic Mode Conversion

   If a complex value is converted to a real value, the result is the real part of the complex value.

   If a real value is converted to a complex value, the result is a complex value that has the real value as the real part and zero as the imaginary part.

Table 1. Arithmetic Base and Scale Conversion

	Before Conversion			
After	Binary Fixed (p,q)	Decimal Fixed(p,q)	Binary Float(p)	Decimal Float(p)
Binary Fixed	(p,q)	(MIN(CEIL(p*3.32) +1,N_1),CEIL(ABS(q) *3.32)*SIGN(q))		
Decimal Fixed	(MIN(CEIL(p/3.32)+1,N_2) CEIL(ABS(q)/3.32) *SIGN (q))	(p,q)		
Binary Float	(MIN(p,N_3))	(MIN(CEIL (p*3.32),N_3))	(p)	(MIN(CEIL (p*3.32),N_3))
Decimal Float	(MIN(CEIL(p/3.32),N_4)	(MIN(p,N_4))	(MIN(CEIL (p/3.32),N_4))	(p)

N_1 is the maximum precision allowed for binary fixed-point data.

N_2 is the maximum precision allowed for decimal fixed-point data.

N_3 is the maximum precision allowed for binary floating-point data.

N_4 is the maximum precision allowed for decimal floating-point data.

2. Integer conversion

   If conversion to integer is specified, as in the evaluation of subscript expressions, the conversion will be to fixed-point binary (x,0). Here x is the total number of positions in the field and depends upon the implementation. The scale factor is zero. Truncation, if necessary, will be toward zero.

3. Arithmetic Base and Scale Conversion

   Table 1 defines the precision resulting from base and scale conversion. CEIL refers to the ceiling of the expression. (The "ceiling" of a number is the smallest integer equal to or greater than the number.)

4. Floating-point to Fixed-point Conversion

   Conversion from floating-point scale to fixed-point scale will occur only when a destination precision is known, as in an assignment to a fixed-point variable. If the destination precision is incapable of holding the floating point value, the result is undefined and the SIZE condition will be raised, if enabled.

<u>Bit-String Operations</u>

Bit-string operations have the following general forms:

```
¬ operand
operand & operand
operand | operand
```

The prefix operation "not" and the infix operations "and" and "or" are specified above. The operands will be converted to bit-string type before the operation is performed. The result will be of bit-string type. If the operands are of different lengths after conversion, the shorter is extended on the right with zeros to the length of the longer. The length of the result will be of this extended length. The result is of varying length if either operand has the VARYING attribute.

The operations are performed on a bit-by-bit basis. As a result of the operations, each bit position has the value defined in the following table:

A	B	not A	not B	A and B	A or B
1	1	0	0	1	1
1	0	0	1	0	1
0	1	1	0	0	1
0	0	1	1	0	0

Examples:

If field A is '010111'B, field B is '111111'B, and field C is '101'B, then

```
     ¬ A yields '101000'B
   C & B yields '101000'B
 A | ¬ C yields '010111'B
¬(¬C|¬B) yields '101111'B
```

For a discussion of how these expressions are evaluated, see "Evaluation of Expressions," in this chapter.

Comparison Operations

Comparison operations have the general form:

operand {<|¬<|<=|=|¬=|>=|>|¬>} operand

There are three types of comparisons:

1. Arithmetic, which involves the comparison of signed numeric values in coded arithmetic form. Conversion of numeric fields will be performed.

2. Character, which involves left-to-right, pair-by-pair comparisons of characters according to the implementation-defined collating sequence. If the operands are of different lengths, the shorter is extended to the right with blanks.

3. Bit, which involves the left-to-right comparison of binary digits. If the strings are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison is a bit string of length one; the value is '1'B if the relationship is true or '0'B if it is false.

Comparison operations always take place between values in common representation. If the operands of a comparison are of different types, the operand of the lower type is converted to conform with the representation of the operand of the higher type. The priority of types is (1) arithmetic (highest), (2) character string, (3) bit string. If one or both of the operands is arithmetic, the operands are converted to the same attributes as those defined for arithmetic operations.

As a result of the conversion, both operands will then be arithmetic or character string, and arithmetic or character comparison will be performed.

Only the operations = and ¬= can be used if either operand is complex.

Only the operators = and ¬= may be used with locator variables, and both operands must be locator variables or a function that returns a locator value.

Concatenation Operations

Concatenation operations have the following general form:

operand||operand

If both operands are of bit-string type, no conversion is performed, and the result is of bit type. In all other cases, the operands are converted where necessary to character-string type before the concatenation is performed, and the result is of character type. The length of the result is the sum of the lengths of the two operands. The result is a varying string if either of the operands has the VARYING attribute.

Examples:

If A is '010111'B, B is '101'B, C is 'XY,Z' and D is 'AA/BB', then

```
   A||B yields '010111101'B
A||A||B yields '010111010111101'B
   C||D yields 'XY,ZAA/BB'
   D||C yields 'AA/BBXY,Z'
```

Type Conversion

Bit String to Character String

The bit 1 becomes the character 1, and the bit 0, the character 0. The length is unchanged. The null bit string becomes the null character string.

Character String to Bit String

The characters 1 and 0 become the bits 1 and 0. The conversion condition will be raised if the character string contains characters other than 0 and 1 in the portion of the string to be converted. The null character string becomes the null bit string.

Character String to Arithmetic

The string for conversion must contain one of the following:

1. [+|-] arithmetic-constant

2. [+|-] real constant {+|-} imaginary-constant

The optionally signed constant or complex expression may be surrounded by an arbitrary number of blanks. However, blanks may not appear between the optional sign and the constant, nor may they precede the central sign in a complex expression. The string must not contain a sterling constant.

The arithmetic value of the constant is converted to the base, scale, mode, and precision that a REAL FIXED DECIMAL value of maximum fixed decimal precision would have been converted to if this had appeared in place of the character string value. A null string gives the value zero.

Bit String to Arithmetic

The bit string is interpreted as an unsigned binary integer, and is converted to the base, scale, mode, and precision that a real fixed binary value of maximum fixed binary precision would have been converted to had it appeared. A null string gives the value 0.

Arithmetic to Character String

The arithmetic value is converted to a character string according to the rules of list-directed output specified in Chapter 7.

Arithmetic to Bit String

The absolute arithmetic value is converted to real then to fixed-point binary, precision (p,0), where p is related to the precision before conversion as follows (with ceilings of expressions used):

BINARY FIXED (r,s)	p = min((N_1,max(r-s,0))
DECIMAL FIXED (r,s)	p = min(N_2,max(CEIL ((r-s)*3.32),0))
BINARY FLOAT (r)	p = min(N_3,r)
DECIMAL FLOAT (r)	p = min(N ,CEIL (r*3.32))

The resulting binary fixed-point value is interpreted as a bit string of length p.

The result of a conversion to fixed-point binary with precision (0,0) is the null bit string.

AGGREGATE EXPRESSIONS

An aggregate expression is an expression involving one or more aggregate operands, i.e. array or structure operands. An aggregate expression is either an array expression or a structure expression. For convenience, array expressions are classified into simple array expressions, whose operands are not structures or arrays of structures, and array of structure expressions. See "The Assignment Statement," in Chapter 8.

Prefix Operators and Aggregate Operands

A prefix operator applied to an aggregate yields a result whose aggregate type is the same as the operand. Thus if A is an array and B is a structure -A is an array expression and -B is a structure expression. The bounds and number of dimensions of an array expression are those of the operand.

Infix Operators and Aggregate Operands

An infix operator applied to two aggregate operands, or to an aggregate operand and a scalar, yields a result whose aggregate type is determined by the operands. The following table gives the aggregate type of the result of an infix operation in terms of the aggregate type of the operands:

Operand 1	Operand 2			
	scalar	simple array	structure	array of structures
scalar	scalar	simple array	structure	array of structures
simple array	simple array	simple array	array of structures	array of structures
structure	structure	array of structures	structure	array of structures
array of structures	array of structures	array of structures	array of structures	array of structures

If both operands are arrays they must have the same bounds and number of dimensions; the result has these common bounds and number of dimensions. If only one operand is an array the result has the bounds and number of dimensions of this array. When structures are involved, they must all have the same structuring.

Built-in Functions with Aggregate Arguments

The built-in functions listed under "Arithmetic Built-in Functions," "Mathematical Built-in Functions," and "String Built-in Functions" in Appendix 1 may be given aggregate expressions in argument positions other than those which must be integer constants. The aggregate type of the result, its bounds and number of dimensions, with n argument positions other than integer constant ones can be obtained by treating the reference as an expression involving these n operands and (n-1) infix operators.

For example, if A is a structure, B is a simple array and C is a scalar.

SIN(A)	is a structure expression
MAX(B,C)	is an array expression
MIN(A,B)	is an array of structures expression

Value of an Aggregate Expression

Aggregate expressions can be used only on the right hand side of an assignment statement, as arguments, and in a data list of a PUT statement.

In an assignment statement the values designated by an aggregate expression are assigned to one or more aggregate target variables. Such an assignment is carried out as a sequence of scalar assignments (see "The Assignment Statement," in Chapter 8). The definition has two major consequences:

1. Array expressions may not yield the results of conventional matrix algebra.

2. When a variable, or part thereof, is specified both as an operand and as a target, the values of the variable when used in the expression may be those assigned earlier in the sequence of scalar assignments.

In other cases no named target variable is available. When passing arguments a dummy variable (the dummy argument) is constructed. The aggregate type of the dummy argument is that specified in the corresponding parameter position of an entry attribute, or if this information is not specified in an entry attribute then the aggregate type is that of the expression itself. The values transmitted to the parameter are determined by assignment of the expression to the dummy argument. The values transmitted by an aggregate expression in an output data list are those which would be assigned to a target variable having the aggregate type of the expression.

EVALUATION OF EXPRESSIONS

In the evaluation of an expression, the priority of operations is as follows.

```
Highest:  ¬,**,prefix +, prefix -
          *, /
          infix +, infix -
          ||
          >=, >, ¬>, ¬=, <, ¬<, <=, =
          &
Lowest:   |
```

Operations within an expression are performed in the order of decreasing priority. For example, in the expression A+B**3, exponentiation is performed before addition. If an expression involves operations of the same priority, the operations ¬, **, prefix +, and prefix - are performed from right to left and all other operations are performed from left to right.

If an expression is enclosed in parentheses, it is treated as a single operand. The parenthesized expression is evaluated before its associated operation is performed. For example, in the expression (A+B**3)/(C*D||E), A will be added to B**3, C*D will be concatenated with E, and then the first of these results will be divided by the second.

Thus, parentheses modify the normal rules of priority.

An implementation may cause evaluation of subscripts, function references, and locator qualifiers in any order that it chooses. This is subject only to the constraint that an operand must be fully evaluated before its value is used in an operation.

The operators + and * are commutative, but not associative, as low-order rounding errors will depend on the order of evaluation of an expression. Thus, A+B+C is not necessarily equal to A+(B+C).

The rules relating to irreducible functions and abnormal data should be noted (see "Abnormality and Irreducibility," in Chapter 10).

ORDER OF THE EVALUATION OF AGGREGATE EXPRESSIONS

Array expressions are evaluated by performing, in turn, a complete scalar evaluation of the expression for each position of the array. The evaluations proceed in row-major order (final subscript varying most rapidly). The result of an evaluation for an earlier position can alter the values of scalar elements for the evaluation of a later position (see Example 1, for "The Assignment Statement," in Chapter 8).

Structure expressions are evaluated by performing a complete scalar evaluation of the expression for each eligible field, in the order in which the fields in the structures are declared. The results of an evaluation for an earlier position can alter the result for the evaluation of a later position.

An identifier appearing in a PL/I program may refer to one of many classes of objects. For example, it may represent a variable referring to a complex number expressed in fixed-point scale with decimal base; it may refer to a file; it may represent a variable referring to a character string; it may represent a statement label or represent a variable referring to a statement label; it may be a variable referring to a pointer or area, etc.

The recognition of an identifier as a particular name is established through declaration of the name.

Those properties that characterize the object represented by the name, and the scope of the name itself, together make up the set of attributes that are to be associated with the name.

There are a number of classes of attributes. These classes and the attributes in each class are described further on in this chapter.

When an identifier is used in a given context in a program, attributes from certain of these attribute-classes must be known in order to assign a unique meaning to the identifier. For example, if an identifier is used as a data variable, the data type must be known; if the data type is arithmetic, the base, scale, mode, and precision must be known.

Examples of Attributes:

CHARACTER (50)--Association of this attribute with an identifier defines the identifier as representing a variable referring to a string 50 characters in length.

FLOAT--Association of this attribute with an identifier defines the identifier as representing a variable referring to arithmetic data, where the data is represented internally in floating-point form.

EXTERNAL--Association of this attribute with an identifier defines the identifier as a name with a certain special scope.

DECLARATIONS

A given identifier is established as a name, which holds throughout a certain scope in the program (see "Scope of Declarations" in this chapter), and a set of attributes may be associated with the name by means of a declaration.

If a declaration is internal to a certain block, then the name is said to be declared in that block.

In a program, a given identifier may be established in different parts of the program as different names. For example, an identifier may represent an arithmetic variable in one part of a program and an entry name in another part. These two parts, of course, cannot overlap.

Each different use of the identifier is established by a different declaration. References to different uses are distinguished by the rules of scope (see "Scope of Declarations").

Declarations may be explicit, contextual, or implicit.

EXPLICIT DECLARATIONS

Explicit declarations are made through use of the DECLARE statement, label prefixes, and specification in a parameter list; by this means, an identifier is established as a name and can be given a certain set of attributes.

Only one DECLARE statement can be used to establish an internal name. However, complementary sets of explicit declarations are permitted:

1. One explicit declaration of an entry name as a statement prefix may be combined with an explicit declaration in a DECLARE statement.
2. One or more explicit declarations in parameter lists may be combined with an explicit declaration in a DECLARE statement.

All declarations of a complementary set must be internal to the same block.

The DECLARE Statement

Function:

The DECLARE statement is a non-executable statement used for the specification of attributes of simple names.

General Format:

```
DECLARE [level] identifier [attribute]...
 [,[level] identifier [attribute]...]...;
```

Syntax rules:

1. Any number of identifiers may be declared as names in one DECLARE statement.
2. Attributes must follow the names to which they refer. (Note that the above format does not show factoring of attributes, which is allowable as explained later).
3. "Level" is a non-zero decimal integer constant. If it is not specified, level 1 is assumed.
4. A DECLARE statement may have a label prefix, but such use does not cause declaration of the identifier as a label constant.
5. A DECLARE statement cannot have a condition prefix.

General Rules:

1. All of the attributes given explicitly for a particular name must be declared together in one DECLARE statement. (Note that for FILE, certain attributes may be specified in an OPEN statement. See Chapter 7, "File Opening and File Attributes.")
2. The following attributes may not be specified more than once for the same name:

   AREA

   BASED

   BIT

   CHARACTER

   DEFINED

   dimension

   ENTRY (parameter attribute list)

   GENERIC

   INITIAL

   LABEL (list)

   LIKE

   OFFSET

   PICTURE

   POSITION

   precision

   RETURNS
3. Attributes of EXTERNAL names, declared in separate blocks and compilations, must not conflict or supply explicit information that was not explicit or implicit in other declarations.

Example:

```
DECLARE JOE FLOAT, JIM FIXED (5,3),
   JACK BIT (10);
```

JOE is declared to be a floating-point scalar variable, JIM a five-position, fixed-point scalar variable with three places to the right of the decimal point, and JACK a scalar variable of ten bits.

Declaration of Structures

The outermost structure is a major structure, and all contained structures are minor structures.

A structure is specified by declaring the major structure name and following it with the names of all contained elements. Each name is preceded by a level number, which is a non-zero decimal integer constant. A major structure is always at level one and all elements contained in a structure (at level *n*) have a level number that is numerically greater than *n*, but they need not necessarily be at level *n*+1, nor need they all have the same level number.

A minor structure at level *n* contains all following items declared with level numbers greater than *n* up to but not including the next item with a level number less than or equal to *n*. A major structure description is terminated by the declaration of another item at level one, by the declaration of an item having no level number, or by the end of a DECLARE statement.

Factoring in DECLARE Statements

Attributes common to several name declarations can be factored to eliminate repeated specification of the same attribute for many identifiers. This factoring is achieved by enclosing the name declarations in parentheses, and following this by the set of attributes which are to apply. Level numbers also may be factored, but in such cases, the level number precedes the parenthesized list of name declarations. Factoring of attributes is permitted only in the DECLARE statement, but not within an ENTRY attribute declaration.

General format:

declare-statement is defined as:

```
    DECLARE declaration-list;
```

where declaration-list is defined as:

```
    declaration [,declaration]...
```

where declaration is defined as:

```
    [integer] {identifier|
     (declaration-list)}
     (dimension-attribute) [attribute...]
```

Examples:

1.
```
DECLARE ((A FIXED, B FLOAT) STATIC,
        C CONTROLLED ) EXTERNAL;
```

   This declaration is equivalent to the following:

```
DECLARE A FIXED STATIC EXTERNAL,
   B FLOAT STATIC EXTERNAL,
   C CONTROLLED EXTERNAL;
```

2.
```
DECLARE 1 A AUTOMATIC,2(B FIXED, C
FLOAT, D CHAR(10));
```

   This declaration is equivalent to the following:

```
DECLARE 1 A AUTOMATIC,
   2 B FIXED,
   2 C FLOAT,
   2 D CHAR(10);
```

Multiple Declarations and Ambiguous References

Two or more declarations of the same identifier, internal to the same block, constitute a <u>multiple declaration</u> of that identifier only if they have identical qualification (including the case of two or more declarations of an identifier at level 1, i.e., scalars or major structures). Multiple declarations are in error.

Reference to a qualified name is always taken to apply to the identifier (for which the reference is valid) declared in the innermost block containing the reference. Within this block, the reference is unambiguous if either of the following is true:

1. The reference gives a valid qualification for one and only one declaration of the identifier.

2. The reference represents the complete qualification of only one declaration of the identifier. The reference is then taken to apply to this identifier.

Otherwise, the reference is ambiguous and in error.

Examples:

```
1.  DECLARE 1 A, 2 C, 2 D, 3 E;
       BEGIN;
       DECLARE 1 A, 2 B, 3 C, 3 E;
       A.C=D.E;
    A.C refers to C in the inner block.
    D.E refers to E in the outer block.

2.  DECLARE  1 A, 2 B, 2 B, 2 C, 3 D, 2 D;
    B has a multiple declaration.
    A.D refers to the second D, since A.D
       is a complete qualification of only
       the second D; the first D would
       have to be referred to as A.C.D or
       C.D.

3.  DECLARE 1 A, 2 B, 3 C, 2 D, 3 C;
    A.C is ambiguous because neither C is
       completely qualified by this ref-
       erence.

4.  DECLARE 1 A, 2 A, 3 A;
        A  refers to the first A.
      A.A  refers to the second A.
    A.A.A  refers to the third A.

5.  DECLARE X; DECLARE 1 Y, 2 X, 3 Z, 3 A,
    2 Y, 3 Z, 3 A;
        X  refers to the first DECLARE
      Y.Z  is ambiguous
    Y.Y.Z  refers to the second Z
    Y.X.Z  refers to the first Z
```

Label Prefixes

A label acting as a prefix to a PROCEDURE or ENTRY statement explicitly declares the identifier as having the ENTRY attribute. If the PROCEDURE or ENTRY statement applies to an external procedure, the attribute EXTERNAL is given, and this declaration is considered to be internal to an imaginary block containing the external procedure. In all other cases, the attribute INTERNAL is given, and the declaration is internal to the block containing the procedure.

A label acting as a prefix to any other statement is an explicit declaration of the identifier as a statement label constant. The declaration is internal to the block containing the statement.

Parameters

The appearance of an identifier in a parameter list of a PROCEDURE or ENTRY statement is an explicit declaration of the identifier as a parameter.

CONTEXTUAL DECLARATIONS

The syntax of PL/I allows unqualified identifiers appearing in certain contexts to be recognized without an explicit declaration. Such contextual declarations will not, however, override any explicit declaration of the same identifier whose scope includes the block containing a statement that might otherwise cause contextual declaration.

Contextual declarations can occur as follows:

1. Pointer. An undeclared identifier can be contextually declared as a pointer variable if it appears:

   a. In parentheses following the keyword BASED in a BASED attribute specification of a DECLARE statement.

   b. In parentheses following the keyword SET in the SET option of an ALLOCATE or LOCATE or READ statement.

   c. As a locator qualifier.

2. Area. An undeclared identifier can be contextually declared as an area variable if it appears in parentheses following the keyword IN in the IN clause of an ALLOCATE or FREE statement or if it appears in parentheses following the keyword OFFSET in an OFFSET declaration, or by its appearance in an OFFSET attribute specification.

3. Task. An undeclared identifier can be contextually declared as a task variable if it appears in parentheses following the keyword TASK in the TASK option of a CALL statement.

4. Event. An undeclared identifier can be contextually declared as an event variable if it appears:

   a. In parentheses following the keyword EVENT in the EVENT option of a statement.

   b. In parentheses following the keyword WAIT in a WAIT statement.

5. Entry. An undeclared identifier that is not a built-in function name can be contextually declared as an entry name if it appears:

   a. Following the keyword CALL in a CALL statement or CALL option of an INITIAL attribute specification.

   b. In a function reference, when followed by an argument list.

6. Built-in. An undeclared identifier that is the same as a built-in function name can be contextually declared with the BUILTIN attribute if it appears followed by an argument list.

7. File. An undeclared identifier can be contextually declared as a file name if it appears:

   a. In the file option of an input or output statement.

   b. In parentheses following one of the input/output condition names.

8. Condition-name. An undeclared identifier can be contextually declared as a condition name if it appears in parentheses following the keyword CONDITION in an ON, SIGNAL, or REVERT statement.

A contextual declaration is treated as if it had been made in the external procedure, even if the reference is made in an internal block. The scope of a contextually declared name is the entire external procedure, except for any internal blocks in which the same identifier is explicitly declared.

IMPLICIT AND BUILT-IN DECLARATIONS

An identifier that is neither explicitly declared nor contextually declared will be declared implicitly as an arithmetic variable or it will be declared as the name of a built-in function.

Attributes assigned by an implicit declaration depend upon the initial (or only) letter of the identifier. An identifier beginning with any of the letters I through N is assigned the attributes BINARY, FIXED, REAL, and default precision by implicit declaration. An identifier beginning with any other letter, including the three alphabetic extenders, is assigned the attributes DECIMAL, FLOAT, REAL, and default precision.

Whenever an identifier is implicitly declared as a variable, the declaration is treated as if it had been made in the external procedure. Even if the reference causing the declaration appears in a contained block, the scope of an implicitly declared name is the entire external procedure, except for internal blocks in which the same identifier is explicitly declared. Note that a contextual declaration occurring anywhere within an external procedure precludes an implicit declaration of that identifier anywhere within the external procedure.

The identifier will be declared as a built-in function if the identifier name is that of a built-in function and the identifier name is nowhere used

1. As a target variable in an assignment statement

2. As the control variable in a DO-statement

3. As the control variable in a repetitive specification within a data list

4. As a receiving field in the data list of a GET statement

ESTABLISHMENT OF DECLARATIONS

The establishment of declarations of names is based on a system of priority, with explicit declarations having the highest priority. It follows a three-step process:

1. Explicit declarations are established, with the scope of each name determined by the block in which the declaration is made.

2. Undeclared identifiers are scanned to determine if their meaning can be recognized contextually (in one of the eight ways described under "Contextual Declarations"). Note that no contextual declaration of an identifier can be made if the identifier lies within the scope of an already established explicit declaration. If any undeclared identifier is recognized contextually, a declaration is generated, with scope established as if the declaration had been made in the external procedure.

3. Following contextual declaration, implicit declarations or declarations as built-in functions are established for all remaining undeclared identifiers, with scope established as if the declaration were made in the external procedure.

ASSIGNMENT OF ATTRIBUTES TO IDENTIFIERS

Names can be given attributes explicitly through DECLARE statements, by occurrences in certain recognizable contexts, and by default rules for identifiers incompletely described by the programmer.

At the time of declaration, all attributes need not be known. For an identifier occurring as a parameter, the characteristic "parameter" is combined with any explicitly declared attributes and/or defaults. Attributes of a file name can be specified in a DECLARE statement, with additional attributes specified in an OPEN statement or implied by the type of operation specified in a data transmission statement that opens the file implicitly. An identifier occurring as an internal entry label is given the attributes INTERNAL ENTRY, which then are also combined with any declared attributes for that identifier, after which defaults are applied.

If an identifier appears in a context that could furnish a contextual declaration of this identifier, and if the contextual reference occurs in the scope of a DECLARE statement declaring the identifier, then the context cannot add any attributes that are not given explicitly or by default in the DECLARE statement.

Application of Default Attributes

Default assumptions are as follows, for the identifier classes indicated:

ENTRY type: EXTERNAL is assumed. IRREDUCIBLE is also assumed unless USES and/or SETS is specified in which case REDUCIBLE is assumed. Scale, base, mode, and precision defaults for the value returned are the same as for arithmetic type given below.

FILE type: A summary of file default attributes appears in "File Opening and File Attributes" in Chapter 7.

TASK type: ABNORMAL and ALIGNED are assumed. Scope and storage class defaults are the same as for Arithmetic type given below.

EVENT type: Defaults are the same as for TASK type.

LABEL type: Range is assumed to be all labels which could be assigned to the variable. NORMAL and ALIGNED are assumed. Scope and storage class defaults are the same as for arithmetic type given below.

Locator type: NORMAL and ALIGNED are assumed. Scope and storage class defaults are the same as for arithmetic type given below.

AREA type: NORMAL and ALIGNED are assumed. Scope and storage class defaults are the same as for arithmetic type given below.

Condition type: EXTERNAL scope is assumed.

String type: NORMAL and UNALIGNED are assumed. Scope and storage class defaults are the same as for arithmetic type given below.

Major Structure type: NORMAL is assumed.

Minor Structure type: NORMAL is assumed. INTERNAL is assumed.

Elementary Structure Element type: NORMAL is assumed. INTERNAL is assumed. If arithmetic type has been indicated, then scale, base, mode, and precision defaults are the same as for arithmetic type given below.

Arithmetic type: If none of scale, base, and mode has been given, then if the identifier starts with any of the letters I - N, FIXED BINARY REAL is assumed; otherwise FLOAT DECIMAL REAL is assumed. If at least one of these has been given, then the remaining defaults are taken from FLOAT, DECIMAL, and REAL. Default precision is implementation defined, dependent on scale and base. NORMAL, INTERNAL, and ALIGNED are assumed. If no storage class is given, then AUTOMATIC is associated with INTERNAL and STATIC with EXTERNAL.

SCOPE OF DECLARATIONS

When a declaration of an identifier is made in a block, there is a certain well-defined region of the program over which this declaration is applicable. This region is called the scope of the declaration or the scope of the name established by the declaration.

The scope of a declaration of an identifier is defined as that block B to which the declaration is internal, but excluding from block B all contained blocks to which another declaration of the same identifier is internal. (Block B may be an imaginary block that is considered to contain the declaration of an external entry name, as discussed under "Label Prefixes.")

Scope of External Names

In general, distinct declarations of the same identifier imply distinct names with distinct non-overlapping scopes. It is possible, however, to establish the same name for distinct declarations of the same identifier by means of the EXTERNAL attribute. The EXTERNAL attribute is defined as follows:

> A declaration of an identifier that declares the identifier as EXTERNAL is called an external declaration for the identifier. All external declarations for the same identifier in a program will be linked and considered as establishing the same name. The scope of this name will be the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must be consistent, since the declarations all involve a single name. For example, it would be an error if the identifier ID were used as an EXTERNAL file name in some READ statement in a program, and in the same program to declare ID as EXTERNAL ENTRY.

The EXTERNAL attribute can be used to communicate between different external procedures or to obtain non-continuous scopes for a name within an external procedure.

An <u>external name</u> is a name that has the scope attribute EXTERNAL. If a name is not external, it is said to be an <u>internal name</u> and has the scope attribute INTERNAL.

The following examples illustrate scope of declarations. The numbers on the left are for reference only, and are not part of the procedure. See Table 2 for an explanation of the scope and use of each name.

Example 1:

```
1  A:  PROCEDURE;
2      DECLARE (X,Z) FLOAT;
           .
           .
           .
3      B:  PROCEDURE (Y);
4          DECLARE Y BIT (6);
5          C:  BEGIN;
6              DECLARE (A,X) FIXED;
                   .
                   .
                   .
7              Y:  RETURN;
               END C;
           END B;
8      D:  PROCEDURE;
9          DECLARE X FILE;
10         Y = Z;
               .
               .
               .
           END D;
       END A;
```

Since entry names of external procedures and file names have the attribute EXTERNAL, the scope of the entry name A and of the file name X above may include parts of other external procedures of the program.

Example 2:

```
   A:  PROCEDURE;
1      DECLARE X EXTERNAL;
           .
           .
           .
       B:  PROCEDURE;
2          DECLARE X FIXED;
               .
               .
               .
           C:  BEGIN;
3              DECLARE X EXTERNAL;
                   .
                   .
                   .
               END C;
           END B;
       END A;
   D:  PROCEDURE;
4      DECLARE X FIXED;
           .
           .
           .
       E:  PROCEDURE;
5          DECLARE X EXTERNAL;
               .
               .
               .
           END F;
       END D;
```

Table 2. Scope and Use of Names in Example 1, for "Scope of External Names"

<u>Reference Line</u>	<u>Name</u>	<u>Use</u>	<u>Scope (by block names)</u>
1	A	external entry name	all of A except C
2	X	floating-point variable	all of A except C and D
2	Z	floating-point variable	all of A
3	B	internal entry name	all of A
4	Y	bit string	all of B except C
5	C	statement label	all of B
6	A	fixed-point variable	all of C
6	X	fixed-point variable	all of C
7	Y	statement label	all of C
8	D	internal entry name	all of A
9	X	file name	all of D
10	Y	floating-point variable	all of A except B

In Example 2, there are five declarations for the identifier X.

Declaration 2 declares X as a fixed-point variable name; its scope is all of block B except block C.

Declaration 4 declares X as another fixed-point variable name, distinct from that of declaration 2; its scope is all of block D except block E.

Declarations 1,3,5 all establish X as a single name; its scope is all of the program except the scopes of declarations 2 and 4.

Basic Rule on Use of Names

A name is said to be known only within its scope. This definition suggests a basic -- and almost self-evident -- rule on the use of names:

> All appearances of an identifier which are intended to represent a given name in a program must lie within the scope of that name.

There are many implications to the above rule. One of the most important is the limitation of transfer of control by the statement GO TO A, where A is a statement label.

The statement GO TO A, internal to a block B, can cause a transfer of control to another statement internal to block B or to a statement in a block containing B, and to no other statement. In particular, it cannot transfer control to any point within a block contained in B.

THE ATTRIBUTES

The attributes are divided into separate classes, as listed in the following paragraphs. Each attribute is described in detail in the "Alphabetic List of Attributes," below.

DATA ATTRIBUTES

Problem Data Attributes

Attributes for problem data are used to describe arithmetic and string variables. Arithmetic variables have attributes that specify the base, scale, mode, and precision of the data items. String variables have attributes that specify whether the variable represents character strings or bit strings and that specify the length to be maintained. The arithmetic data attributes are:

BINARY|DECIMAL

FIXED|FLOAT

REAL|COMPLEX

(precision)

PICTURE

The string data attributes are:

BIT|CHARACTER

(length)

VARYING

PICTURE

Program Control Data Attributes

Attributes for program control data specify that the associated name is to be used by the programmer to control the execution of this program. The program control data attributes are:

LABEL

TASK

EVENT

CELL

AREA

POINTER

OFFSET

Other Attributes of Data

The INITIAL, DEFINED, ALIGNED, UNALIGNED, storage class, and scope attributes can be declared for both problem data and program control data.

Other attributes apply only to data aggregates. For array variables, the dimension attribute specifies the number of dimensions and the bounds of an array. The LIKE attribute specifies that the structure variable being declared is to have the same structuring as the structure of the name following the attribute LIKE. The SECONDARY attribute specifies that certain data does not require efficient storage.

ENTRY NAME ATTRIBUTES

The entry name attributes identify the name being declared as an entry name and describe features of the associated entry point. For example, the attribute BUILTIN specifies that the reference to the associated name within the scope of the declaration is interpreted as a reference to the built-in function or pseudo-variable of the same name. The entry name attributes are:

ENTRY

RETURNS

GENERIC

BUILTIN

FILE ATTRIBUTES

The file description attributes establish an identifier as a file name and describe characteristics for that file, e.g., how the data of the file is to be transmitted, whether records of a file are to be buffered. If the same file name is declared in more than one external procedure, the declarations must not conflict, unless one is declared with the INTERNAL attribute.

The file attributes are:

FILE

STREAM|RECORD

INPUT|OUTPUT|UPDATE

PRINT

SEQUENTIAL|DIRECT

BUFFERED|UNBUFFERED

BACKWARDS

ENVIRONMENT(option-list)

KEYED

EXCLUSIVE

Note that file description attributes, except for the ENVIRONMENT attribute, can be specified as options in the option list of the OPEN statement.

OPTIMIZATION ATTRIBUTES

The optimization attributes provide information to the compiler to allow (or prevent) optimization of certain portions of the compiled program. They specify the way in which data may be altered and the behavior of procedures when they are invoked. The optimization attributes are:

ABNORMAL|NORMAL

IRREDUCIBLE|REDUCIBLE

SETS (item-list)

USES (item-list)

In the absence of any information to the contrary, the following assumptions are made:

| 1. All entry names are irreducible.

2. All variables are normal.

A variable is said to be abnormal if its value may be altered or otherwise accessed without an explicit indication. Thus, for example, the appearance of a variable name on the left side of an assignment statement, in the data list specification of a GET statement, or as an argument to an irreducible function or procedure (see below) indicates a predictable situation where the variable may change its value. However, when the variable is subject to change by the occurrence of an ON-condition, or if it is subject to change in a procedure invoked with the TASK option (see "Asynchronous Operations and Tasks"), then there is no way to predict the point at which the change in value will occur or, in fact, if it will occur.

Such possibilities cannot always be recognized contextually. Furthermore, if a portion of a source program contains several references to such a variable, the order in which the indicated operations are executed becomes significant. (For example, if B is abnormal, the expression B + B is not necessarily equivalent to the expression 2 * B.)

The implication is that the programmer expects the operation to be performed in a particular order. Such variables must therefore be declared ABNORMAL, to inhibit the optimization of such portions of a source program.

If a function is invoked several times with the same arguments, a compiler may or may not be able to invoke the function once and then, in subsequent references, simply use the value returned by the first invocation. The "irreducibility" of a procedure determines whether the number of times it is invoked may be reduced in this way without altering the results of the program. A procedure is either completely irreducible, definitively irreducible, or reducible.

A procedure is completely irreducible if it, or any of its dynamically descendant blocks, does any of the following:

1. Returns different function values for identical argument values
2. Maintains any kind of history
3. Performs input or output operations
4. Returns control from the procedure by means of a GO TO statement

If any such cases apply, each function reference to the procedure must be evaluated. The IRREDUCIBLE attribute is used to describe such procedures; it must either be given explicitly or obtained by default. The additional specification of USES and SETS is allowed, but will not cause the procedure to be recognized as definitively irreducible.

Provided a procedure is not completely irreducible, it is definitively irreducible if it, or any of its dynamically descendant blocks, accesses, modifies, allocates, or frees any of its arguments or any generation of a variable known in the invoking block. These actions of the procedure may be defined by the USES and SETS attributes. Provided these attributes are specified and the procedure is declared REDUCIBLE or is REDUCIBLE by default, the procedure is recognized as being definitively irreducible. The number of invocations of such a procedure with identical arguments may be reduced provided the following conditions are satisfied:

1. No variable specified in the USES attribute is also specified in the SETS attribute
2. No variable mentioned in the USES and/or SETS attribute has its value changed between the function references

When irreducibility is specified, whether it be complete or definitive, the order of evaluation of expressions becomes significant. Hence, the results of a program in which irreducible functions are invoked may depend on the implementation.

SCOPE ATTRIBUTES

The scope attributes are used to specify whether or not a name may be known in another external procedure. The scope attributes are:

INTERNAL|EXTERNAL

All external declarations for the same identifier in a program are linked as declarations of the same name. The scope of this name is the union of the scopes of all the external declarations for this identifier.

In all of the external declarations for the same identifier, the attributes declared must be consistent, since the declarations all involve a single name. For example, it would be an error if the identifier ID were declared as an EXTERNAL file name in one block and as an EXTERNAL entry name in another block in the same program.

The INTERNAL attribute specifies that the declared name is known only in the declaring block and its contained blocks.

The same identifier may be declared with the INTERNAL attribute in more than one block without regard to whether the attributes given in one block are consistent with the attributes given in another block, since each such declaration establishes a different name.

STORAGE CLASS ATTRIBUTES

The storage class attributes are used to specify the type of storage for a data variable. The storage class attributes are:

STATIC

AUTOMATIC

CONTROLLED

BASED

ALPHABETIC LIST OF ATTRIBUTES

Following are detailed descriptions of the attributes, listed in alphabetic order. Alternative attributes are discussed together, with the discussion listed in the alphabetic location of the attribute whose name is the first in alphabetic order. A cross-reference to the combined discussion appears wherever an alternative appears in the alphabetic listing.

ABNORMAL and NORMAL (Optimization Attributes)

The ABNORMAL and NORMAL attributes specify the ways in which values of variables may be altered.

The NORMAL attribute specifies that the value of a variable will not be changed except through normal assignments that can be predicted. Consequently, the value need not necessarily be accessed each time the variable is referred to.

The ABNORMAL attribute specifies that the value of a variable may be changed at an unpredictable time. Consequently, the value must be accessed each time the variable is referred to. A variable should be declared ABNORMAL if its value might be changed in an on-unit or by references in more than one task.

General format:

    ABNORMAL|NORMAL

General rules:

1. If any component of a structure, either a scalar variable or a minor structure, is declared ABNORMAL, no containing structure name, nor the name of the major structure can be explicitly declared NORMAL. However, contained components of an ABNORMAL structure can be declared with the NORMAL attribute.

2. A structure explicitly declared with the NORMAL attribute cannot contain abnormal components.

Assumptions:

NORMAL is the default. Variables are assumed to be NORMAL unless they are components of a structure declared to be ABNORMAL; such components are assumed to be ABNORMAL unless they are explicitly declared NORMAL. Each component of a structure that has been explicitly declared NORMAL will be given the NORMAL attribute by default. Each ABNORMAL component of a structure will cause its containing components to be ABNORMAL by default. Any structure component that has not been given a NORMAL or ABNORMAL attribute, either explicitly or by default, will be NORMAL by default.

ALIGNED and UNALIGNED (Data Attributes)

The ALIGNED and UNALIGNED attributes specify the arrangement of data elements in storage to provide speed of access or storage economy respectively.

ALIGNED and UNALIGNED are element data attributes, but, syntactically, either may also be applied to any aggregate. This is semantically equivalent to the application of the attribute to all contained elements of the aggregate which are not explicitly declared with the ALIGNED or UNALIGNED attribute.

General format:

    ALIGNED|UNALIGNED

General rules:

1. Application of either attribute to an aggregate affects the contained members, unless any member is explicitly declared otherwise. Thus application of either attribute to a substructure affects the contained members and overrides an ALIGNED or UNALIGNED attribute that may have been implicitly applied to those members by having been specified for the containing structure.

2. The ALIGNED and UNALIGNED attributes are applied by default at element level. The default for bit-class and character-class data is UNALIGNED, and for all other types of data it is ALIGNED.

3. For string overlay defining, all the elements of the defined item must have the UNALIGNED attribute, as must those of the base item covered by the range of defining, i.e., from its beginning for a length equal to the length of the item plus the value of the starting position minus one.

4. For simple and iSUB defining, the attributes ALIGNED and UNALIGNED must agree between corresponding elements of the defined item and the base.

5. The ALIGNED and UNALIGNED attributes of an argument in a procedure invocation must match the attributes of the corresponding parameter. If the attributes of the orginal argument do not match those of the corresponding parameter in an ENTRY attribute declaration, a dummy argument is created with the attributes specified in the ENTRY attribute declaration, and the original argument is assigned to it.

6. If a BASED variable is used to access a generation of another variable, then the ALIGNED and UNALIGNED attributes of the accessed variable and the BASED variable must agree.

7. For all operators and built-in functions, the default for ALIGNED or UNALIGNED is applicable to the elements of the result.

8. Constants take the default for ALIGNED or UNALIGNED.

AREA (Program Control Data Attribute)

The AREA attribute defines storage that, on allocation, is to be reserved for the allocation of based variables. Storage thus reserved can be allocated to and freed from based variables by naming the area variable in the IN option of the ALLOCATE and FREE statements. Storage that has been freed can be subsequently reallocated to a based variable.

General format:

    AREA [(size)]

Syntax rule:

The "size" can be an expression or an asterisk.

General rules:

1. The area size for areas that are not of static storage class is given by an expression which is converted to an integer when the area is allocated. It is used in an implementation-defined way to indicate the amount of storage to be reserved.

2. The size for areas of static storage class must be specified as a decimal integer constant.

3. An asterisk may be used to specify the size if the area variable being declared is controlled or is a parameter. In the case of a controlled area variable that is declared with an asterisk, the size must be specified in the ALLOCATE statement used to allocate the area. In the case of a parameter that is declared with an asterisk, the size is inherited from the argument.

4. The AREA condition is raised if an attempt is made to allocate a based variable in an area that does not contain sufficient free storage for the allocation.

5. Data of the area type cannot be converted to any other type; an area can be assigned to an area variable only.

6. During execution, the state of the storage allocated for an area depends only on the allocations made and freed in the area; it does not depend on the size of the area. This state is represented by the significant allocations made in the area. When an area is allocated, it contains no significant allocations; its value is identical to the EMPTY built-in function. An allocation, A, made in an area is significant at some given time if it has not been freed by that time. If it has been freed by that time, it is significant only if a subsequent significant allocation was made before A was freed.

7. No operators can be applied to area variables. An area expression is either a reference to an area variable or a reference to a function returning data of area type.

8. Only the INITIAL CALL form of the INITIAL attribute is allowed with area variables. Since area variables are effectively initialized to the value of the EMPTY built-in function, only one alternative of a cell can be, or can contain, data of area type.

9. An area may have the DEFINED attribute. Only simple and iSUB defining are allowed. The base must have the same size as the defined area.

10. Area data may be transmitted in RECORD I/O; it maintains its validity. Area data cannot be transmitted by STREAM I/O.

Assumptions

1. If the size is omitted, an implementation-defined default value is supplied.

2. An area variable can be contextually declared by its appearance in an

OFFSET attribute specification or in an IN option.

AUTOMATIC, STATIC, CONTROLLED and BASED (Storage Class Attributes)

The storage class attributes are used to specify the type of storage allocation to be used for data variables.

AUTOMATIC specifies that storage is to be allocated upon each entry to the block to which the storage declaration is internal. The storage is released upon exit from the block. If the block is a procedure that is invoked recursively, the previously allocated storage is "pushed down" upon entry; the latest allocation of storage is "popped up" upon termination of each generation of the recursive procedure.

STATIC specifies that storage is to be allocated when the program is loaded and is not to be released until program execution has been completed.

CONTROLLED specifies that full control will be maintained by the programmer over the allocation and freeing of storage by means of the ALLOCATE and FREE statements. Multiple allocations of the same controlled variable, without intervening freeing, will cause stacking of generations of the variable.

BASED, like CONTROLLED, specifies that full control over allocation and freeing will be maintained by the programmer. However, the separate generations are not stacked; each may be accessed by a pointer value that identifies the generation and is used as a locator qualifier applied to the based variable. A based variable can be used to identify data of any storage class by associating the based variable name with a locator qualifier that points to that data. Based variables can be allocated and freed by use of the ALLOCATE and FREE statements. Such allocations are not stacked. Any generation is available as long as it remains in an allocated state.

General format:

```
{STATIC                                    }
{AUTOMATIC                                 }
{CONTROLLED                                }
{BASED[(scalar-locator-expression)]        }
```

General rules:

1. Automatic and based variables can have internal scope only. Static and controlled variables may have either internal or external scope.

2. Storage class attributes cannot be specified for entry names, file names, members of structures, or DEFINED data items.

3. STATIC, BASED, and AUTOMATIC attributes cannot be specified for parameters.

4. Variables declared with adjustable lengths and dimensions cannot have the STATIC attribute.

5. For a structure variable, a storage class attribute can be given only for the major structure name. The attribute then applies to all elements of the structure or to the entire array of structures. If the CONTROLLED or BASED attribute is given to a structure, only the major structure and not the elements can be allocated and freed.

6. If, during evaluation of an expression, a controlled or based variable is allocated or freed, the result of the statement depends upon the implementation in those cases in which the variable is used elsewhere in the statement.

7. The following rules govern the use of based variables:

   a. If no locator expression is specified, any reference to the based variable must have an explicit locator qualifier. This does not apply to a based variable that is the object of a REFER option or that is to be allocated through the use of an ALLOCATE or LOCATE statement.

   b. A reference to a based variable without an explicit locator qualifier is implicitly qualified by the locator expression in the BASED attribute specification in the DECLARE statement for that based variable. Identifiers in this implicit qualification are those of the names in the declaring block. Expressions occuring in this implicit qualifier are evaluated in the current environment of the declaring block with enabling of conditions as the enabling of conditions exists at the point of reference. Consider the following example:

```
    DECLARE B BASED (P(I)),
      P(3) POINTER;
         .
         .
         .
    BEGIN;
    DECLARE P POINTER, I;
         .
         .
         .
L: B = X;
```

The statement B=X has the same effect as:

```
      P(I)->B=X;
```

Where both P and I are the names known in the outer block, not those declared in the begin block. Conditions enabled at L are used when P(I) is evaluated.

c. When a reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated locator variable identifies the generation of data. If the reference is to a component of a based structure, a second, temporary locator variable is created to determine the location of the component in relation to the beginning of the structure.

d. Array bounds and string lengths declared with the based variable are evaluated dynamically with each reference to the based variable. Therefore, the asterisk notation for dimensions and lengths is not permitted. A reference to a component of a based structure causes evaluation of sufficient elements of the structure to determine the position of the component.

e. When a based variable is allocated using the ALLOCATE or LOCATE statement, expressions for bounds, lengths, and area sizes are evaluated at the time of allocation.

f. The REFER option can be used to create structures that contain self-defining data. It may be used in a DECLARE statement to specify a bound of an array, the length of a string, or the size of an area. The REFER option has the following form:

```
    expression REFER(unsubscripted-
        scalar-variable)
```

The "unsubscripted-scalar-variable," which is the object of the REFER option, must be the name of a preceding scalar member of the structure containing the REFER option.

Upon allocation of a structure containing one or more REFER options, all expressions for bounds, string lengths, and area sizes are evaluated (in any order), a new generation of the structure is then allocated, and the relevant locator variable is assigned a value to identify this generation. Initialization is then done (in any order) for the new generation of variables that are objects of the REFER options, using the value obtained for each from the expression appearing in its respective REFER option.

In a reference specifying some generation of a based variable, some of whose bounds, lenghts, and sizes are specified by REFER options, these values are taken from the values of those variables in the generation referred to, that are objects of the REFER options.

Note: The unsubscripted variable that is the object of the REFER option differs from other based variables in that when a reference is made to it, the implied pointer from the based variable is not used, but the reference is always to that generation of the structure that is currently being accessed or allocated.

g. The EXTERNAL attribute cannot be specified for a based variable.

h. The VARYING attribute cannot be specified for a based variable.

i. The INITIAL attribute may be specified for a based variable. The values are used only upon explicit allocation of the based variable with an ALLOCATE or LOCATE statement. If the REFER option appears in a structure for which any element has the INITIAL attribute, initialization specified by the INITIAL attribute is done after contained variables named in all REFER options have been assigned their proper values.

j. A based variable cannot appear in the item list of a CHECK condition prefix, nor in a data-directed data list.

k. Whenever a based variable containing arrays, strings, or areas is passed as an argument, dimensions, lengths, and sizes are determined at the time the argument is passed and remain fixed throughout execution of the invoked block.

Assumptions:

1. If no storage class attribute is specified and the scope is internal, AUTOMATIC is assumed.

2. If no storage class attribute is specified and the scope is external, STATIC is assumed.

3. If neither the storage class nor the scope attribute is specified, AUTOMATIC is assumed.

4. An undeclared identifier appearing in parentheses following the keyword BASED in the BASED attribute specification is contextually declared with the POINTER and AUTOMATIC attributes.

BACKWARDS (File Description Attribute)

The BACKWARDS attribute specifies that the records of a SEQUENTIAL INPUT file are to be accessed in reverse order, i.e., from the last record to the first record.

General format:

```
    BACKWARDS
```

General rule:

The BACKWARDS attribute applies to RECORD files only; that is, it conflicts with the STREAM attribute. It implies RECORD and SEQUENTIAL.

BASED (Storage Class Attibute)

See AUTOMATIC.

BINARY and DECIMAL (Arithmetic Data Attributes)

The BINARY and DECIMAL attributes specify the base of the data items represented by an arithmetic variable as either binary or decimal.

General format:

```
    BINARY|DECIMAL
```

General rule:

The BINARY or DECIMAL attribute cannot be specified with the PICTURE attribute.

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the ABNORMAL, NORMAL, DEFINED, SECONDARY, INITIAL, ALIGNED, UNALIGNED, dimension, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are REAL FIXED BINARY with default precision. For identifiers beginning with any other alphabetic character, the default attributes are REAL FLOAT DECIMAL with default precision. If FIXED or FLOAT and/or REAL or COMPLEX are declared, then DECIMAL is assumed.

BIT and CHARACTER (String Attributes)

The BIT and CHARACTER attributes are used to specify string variables. The BIT attribute specifies a bit string. The CHARACTER attribute specifies a character string. The length attribute for the string must also be specified.

General format:

$$\begin{Bmatrix}\text{BIT}\\ \text{CHARACTER}\end{Bmatrix}\ \text{(length) [VARYING]}$$

General rules:

1. The length attribute specifies the length of a fixed-length string or the maximum length of a varying-length string.

2. The VARYING attribute specifies that the variable is to represent varying-length strings, in which case, length specifies the maximum length. The current length at any time is the length of the current value. VARYING may appear anywhere in the declaration of the string, and it may be factored. VARYING cannot be specified for defined or based variables.

3. The length attribute must immediately follow the CHARACTER or BIT attribute at the same factoring level with or without intervening blanks.

4. The length attribute may be specified by an expression or an asterisk.

   If the length specification is an expression, it is converted to an integer when storage is allocated for the variable.

   The asterisk notation can be used for specification of parameters and controlled variables. In the case of parameters other than controlled parameters, it indicates that the length is to be that of the argument; otherwise a decimal integer constant is required. In the case of controlled variables, it indicates that the length is to be specified when the variable is allocated. For based variables, the asterisk notation cannot be used, but the REFER option of the BASED attribute can be used to specify length at allocation time.

5. If a string has the STATIC attribute, the length attribute must be a decimal integer constant.

6. The BIT, CHARACTER, and VARYING attributes cannot be specified with the PICTURE attribute.

7. The PICTURE attribute can be used instead of CHARACTER to declare a fixed-length string variable (see the PICTURE attribute).

8. All of the string attributes must be declared explicitly unless the PICTURE attribute is used. There are no defaults for string data.

BUFFERED and UNBUFFERED (File Description Attributes)

The BUFFERED attribute specifies that during transmission to and from external storage each record of a SEQUENTIAL RECORD file must pass through intermediate storage buffers.

The UNBUFFERED attribute specifies that such records need not pass through buffers. It does not, however, specify that they must not.

General format:

    BUFFERED|UNBUFFERED

General rule:

The BUFFERED and UNBUFFERED attributes can be specified for SEQUENTIAL RECORD files only.

Assumption:

Default is BUFFERED.

BUILTIN (Entry Attribute)

The BUILTIN attribute specifies that any reference to the associated name within the scope of the declaration is to be interpreted as a reference to the built-in function or pseudo-variable of the same name.

General format:

    BUILTIN

General rules:

1. BUILTIN is used to refer to a built-in function or pseudo-variable in a block that is contained in another block in which the same identifier has been declared to have another meaning.

2. If the BUILTIN attribute is declared for an entry name, the entry name can have no other attributes.

3. The BUILTIN attribute cannot be declared for parameters.

CELL (Program Control Data Attribute)

Function:

The CELL attribute establishes the associated identifier as a cell and specifies that each declaration in the alternative list will occupy the same storage as the other alternative declarations in the list. It differs from the DEFINED attribute in that it provides storage equivalence (i.e., different data declarations occupying the same storage), whereas the DEFINED attribute provides data equivalence (i.e., different ways of referring to the same data).

General format:

    CELL alternative-list

Syntax rules:

1. The alternative list should contain the data declarations of at least two alternatives. This declaration of a cell is the same as the declaration for a structure except that the CELL attribute is specified for the first name.

2. Each alternative declaration must be preceded by a level number, which must be numerically greater than the level number of the cell identifier.

3. The cell identifier may be given other attributes. These attributes may be specified either before or after the keyword CELL but not after the alternative list. The only other attributes that a cell identifier may have are as follows:

   a. The dimension attribute

   b. ABNORMAL or NORMAL

   c. Any of the storage class attributes

   d. EXTERNAL or INTERNAL

   e. SECONDARY

   Note that c, d, and e may be given only for a cell at level 1.

General rules:

1. Each alternative may have any of the attributes that a structure component may have.

2. Each alternative is qualified by the name of the cell to which it belongs and may be referred to as such.

3. Any dimension that a cell identifier has been given is inherited by the alternatives of that cell.

4. Only one alternative may be active at one time. In other words, at any one point in time, only one alternative of a cell can contain a value. An assignment to one alternative effectively deactivates the previously active alternative.

5. Only one alternative of a cell may have the INITIAL attribute.

6. A cell may appear only in DECLARE, ALLOCATE, and FREE statements, as well as in the context of arguments and parameters.

7. Only one AREA alternative is allowed for a single cell variable, and if an alternative contains an area, no other alternative can have the INITIAL attribute.

Examples:

1.
```
DECLARE 1 AAA,
              2 BBB CELL,
                 3 U POINTER,
                 3 V FLOAT (12),
                 3 W CELL,
                    4 XX CHARACTER (20),
                    4 YY BIT (100),
              2 CCC CHARACTER (5),
              2 DDD (20) CELL,
                 3 EE BIT (5),
                 3 FF CHARACTER (1);
```

The above example describes a structure AAA whose components are as follows:

   a. BBB, a cell whose alternatives are the pointer variable U, the floating-point variable V, and another cell, W. The cell W, in turn, contains two alternatives: the character string XX and the bit string YY.

   b. CCC, a character string.

   c. DDD, an array of 20 elements, each of which is a cell having two alternatives: bit string EE and character string FF. Note that DDD(10).EE and EE(10) are references to the same alternative; namely, the bit string alternative for the tenth cell in DDD.

2.
```
DECLARE 1 A CELL CONTROLLED,
               2 B FLOAT (8),
               2 C FIXED (10);
          .
          .
          .
ALLOCATE A;
          .
          .
          .
          FREE A;
          .
          .
          .
```

In this example, A is a cell whose storage is allocated and freed by the use of the ALLOCATE and FREE statements. During the time that A remains allocated, its alternatives, B and C, are available for use.

CHARACTER (String Attribute)

See BIT.

COMPLEX and REAL (Arithmetic Data Attributes)

The COMPLEX and REAL attributes are used to specify the mode of an arithmetic variable. REAL specifies that the data items represented by the variable are to be real numbers. COMPLEX specifies that the data items represented by the variable are to be complex numbers, that is, each data item is a pair: the first member is a real number and the second member an imaginary number.

General format:

```
REAL|COMPLEX
```

General rule:

1. If a numeric character variable is to represent complex values, the COMPLEX attribute must be specified with the PICTURE attribute. The COMPLEX or REAL attribute is the only other arithmetic or string data attribute that can be specified with the PICTURE attribute.
2. A single precision attribute applies to a complex variable (unless it is declared with the PICTURE attribute). It specifies the precision of both the real and the imaginary parts.

Assumption:

Default is REAL.

CONTROLLED (Storage Class Attribute)

See AUTOMATIC.

DECIMAL (Arithmetic Data Attribute)

See BINARY.

DEFINED (Data Attribute)

The DEFINED attribute specifies that the level one scalar, array, or structure data is to occupy some or all of the storage assigned to the base item specified in the attribute.

General format:

```
DEFINED base-item
```

Rules for defining:

1. The INITIAL, the storage class, and the EXTERNAL attributes must not be specified for the defined item, nor may the defined item be a parameter. Neither the defined item nor the base item may contain VARYING strings. The defined item is internal by default.
2. The base item is a (possibly subscripted) scalar, array, or structure. It must not have the based attribute or the defined attribute.
3. In references to defined data, the bounds and string lengths of the defined data are used to determine whether the STRINGRANGE and SUBSCRIPTRANGE conditions occur.

There are three types of defining, simple defining, iSUB defining, and string overlay defining.

If the POSITION attribute is specified for the defined item, string overlay defining is in effect; in this case the base item must not contain references to iSUB variables. If the subscripts specified in the base item contain any references to iSUB variables, iSUB defining is in effect. If neither iSUB variables nor the POSITION attribute is present, then simple defining is in effect if the base item and defined item match according to the criteria given below; otherwise, string overlay defining is in effect.

A base item and a defined item match if the base item when passed as an argument would match a parameter which had the attributes of the defined item (apart from the defined attribute). For this purpose, the parameter is assumed to have all bounds, string lengths, and area sizes specified by asterisks.

Simple Defining

Simple defining allows a (possibly subscripted) scalar, array, or structure item to be accessed by a different name. The attributes ALIGNED and UNALIGNED must agree between corresponding elements of the defined item and base. Array bounds and string lengths associated with the defined item may differ from those of the base item, although they are subject to certain constraints given below.

1. Corresponding to any simple defined reference, there is an equivalent reference to the base item given in the DEFINED attribute of the defined item. The qualified name in this equivalent reference is the name of the base item; if the defined reference was qualified, the equivalent reference is further qualified by those identifiers in the declaration of the base item which correspond to the qualifying identifiers in the defined reference. If the base item names an array the equivalent reference contains a subscript corresponding to each dimension in the array. The ith subscript in the equivalent reference is the ith subscript specified in the base item, unless an asterisk is specified for the base item in the DEFINED attribute specification. Wherever an asterisk appears, it indicates that the subscript to be used in the equivalent reference is the corresponding subscript of the reference to the defined item.

2. The range specified by a bound pair of a defined array must equal or be contained within the range specified by the corresponding bound pair of the base array.

3. The length of a simple defined string must not be greater than the length of the corresponding base string.

4. The size of a simple defined area must be equal to the size of the corresponding base area.

   Example:

   ```
   DECLARE A(10),1 X(M,N),2 Y,2 Z,
           C DEFINED A(3),
           1 E(M/2) DEFINED X(*,I),2 F,
             2 G;
   ```

C refers to A(3), E.F(3) refers to X.Y(3,I).

iSUB Defining

The use of iSUB defining allows a transformation to be applied to the subscripts of a defined reference to designate a chosen element of the base array. If the defined reference does not specify some subscript expression, the transformation is applied to the subscripts generated during the evaluation of the aggregate expression or aggregate assignment which contains the reference. The defined item and base items may be arrays of structures.

The subscripts in the base item in the DEFINED attribute make one or more references to the dummy iSUB variables; i is a decimal integer constant in the range 1 to n where n is the number of dimensions in the defined array. The number of subscripts in the base item must be equal to the number of declared dimensions of the base array; subscript positions must not be specified by asterisks.

Corresponding to a subscripted iSUB-defined reference is an equivalent subscripted reference to an element of the base array. The qualified name part is derived in the way used for simple-defined references. However, the subscript list is derived differently. The jth subscript in the equivalent reference is the jth subscript in the base item, after each iSUB variable has been replaced by the integer value of the ith subscript in the defined reference.

The attributes of the base array and of the defined array must obey the rules for valid simple defining.

An array reference to an iSUB array must not be passed as an argument, unless a dummy is created. Scalar references to iSUB defined arrays may be passed without the creation of a dummy.

Within the expressions in a base item, iSUB variables are treated as fixed binary variables with the precision given by the conversion rules.

Example:

```
    DECLARE X(10,10),Y(5)
            DEFINED X(2*1SUB,2*1SUB);
```

The array Y refers to the even elements of the diagonal of X. Thus Y(1) refers to X(2,2), Y(2) to X(4,4), etc.

String Overlay Defining

String overlay defining is applicable only to string and pictured data. It enables some or all of the storage associated with a variable to be accessed using any suitable string or pictured scalar or an aggregate of string and pictured data.

The POSITION attribute can be used to specify the bit or character within the base item at which the defined item is to begin. Its format is:

```
    POSITION(decimal-integer-constant)
```

It may appear anywhere within the declaration of the level-one name of the defined item. If it is omitted POSITION(1) is assumed. The number of bits or characters in the defined item, plus n-1 where n is the decimal integer constant in the POSITION attribute, must be not greater than the number of bits or characters in the base item.

The defined item and the base item must both be of bit class or both be of character class. The bit class consists of:

a. Unaligned fixed-length bit strings

b. Unaligned binary numeric data

c. Aggregates consisting of items a and b

The composition of the character class is:

a. Unaligned fixed-length character strings

b. Unaligned decimal numeric data

c. Unaligned character-string pictured data

d. Aggregates consisting of items a, b, and c

All the elements of the base item covered by the range of defining and all the elements of the defined item must have the UNALIGNED attribute.

The base item cannot be an aggregate parameter, nor can it be an interleaved array. An interleaved array is an array whose associated storage contains gaps occupied by other fields; an array is interleaved if, when written in cross-section notation, it has an asterisk to the right of any subscript expression or has no asterisk corresponding to an array of structures which contains the array.

Example:

```
DECLARE A CHARACTER(10),
        B(10) CHARACTER(1) DEFINED A;
B = '0';
```

The assignment to B sets each character in A to '0'.

Order of Evaluation

Evaluation proceeds as follows:

1. The array bounds, string lengths, and area sizes of a defined item are evaluated upon entry to the block in which the item is declared.

2. A defined reference is treated as a reference to some or all of that generation of its base item that is available at the point of reference. When a defined item is passed as an argument without creation of a dummy, the corresponding parameter refers to the relevant part of that generation of the base item that is available when the argument is passed; reallocation of the base within the called procedure will not affect the meaning of the parameter.

3. In a reference to a defined item, all subscripts in the reference are evaluated and converted to integer before any of the subscripts in the base item are evaluated. Expressions in the base item are then evaluated in the current environment of the block containing the declaration of the defined item; names used in the base item are interpreted in the block containing the declaration of the defined item.

Dimension (Array Attribute)

The dimension attribute specifies the number of dimensions of an array and the bounds of each dimension. The dimension attribute either specifies the bounds (only the upper bound or both the upper and lower bounds) or indicates, by use of an asterisk, that the actual bounds for the array are to be taken from elsewhere.

General format:

```
(bound [,bound]...)
```

where "bound" is:

```
{[lower-bound:] upper-bound}|*
```

and "upper-bound" and "lower-bound" are element expressions.

General rules:

1. The number of bounds specifications indicates the number of dimensions in the array unless the variable being declared is contained in an array of structures, in which case it inherits dimensions from the containing structure.

2. The bounds specification indicates the bounds as follows:

   a. If only the upper bound is given, the lower bound is assumed to be 1.

   b. On allocation of storage, the lower bound must be less than or equal to the upper bound.

c. An asterisk specifies that the actual bounds are to be specified in an ALLOCATE statement, if the variable is controlled, or are to be taken from the argument (other than for a controlled parameter), if the variable is a parameter. The asterisk notation can be used only for parameters and CONTROLLED variables.

3. Bounds that are expressions are evaluated and converted to integer data when storage is allocated for the array. Bounds in a parameter attribute list that are specified by expressions are evaluated in the prologue of the block containing the entry attribute that specifies them; this does not apply to the bounds in a controlled parameter attribute list, which are never evaluated. For simple parameters, bounds can be only optionally signed decimal integer constants or asterisks.

4. The bounds of arrays declared STATIC must be optionally signed decimal integer constants.

5. The dimension attribute must immediately follow the array name (or the parenthesized list of names, if it is being factored). Intervening blanks are optional.

6. The asterisk notation cannot be used for based variables, but the REFER option can be used to specify a bound at the time of allocation.

DIRECT and SEQUENTIAL (File Description Attributes)

The DIRECT and SEQUENTIAL attributes specify the manner in which the records of a RECORD file are to be accessed. SEQUENTIAL specifies that the records are to be accessed according to their logical sequence in the data set. DIRECT specifies that the records of the file are to be accessed by use of a key. Each record of a direct file must, therefore, have a key associated with it. Either of these attributes implies the RECORD attribute.

Note that SEQUENTIAL and DIRECT specify only the current usage of the file; they do not specify physical properties of the data set associated with the file. A SEQUENTIAL file may actually have keys recorded with the data.

General format:

```
SEQUENTIAL|DIRECT
```

General rules:

1. DIRECT files must also have the KEYED attribute which is implied by DIRECT. SEQUENTIAL files may or may not have the KEYED attribute.

2. The DIRECT and SEQUENTIAL attributes cannot be specified with the STREAM attribute.

Assumptions:

1. Default is SEQUENTIAL for RECORD files.

2. If a file is implicitly opened by an UNLOCK statement, DIRECT is assumed; if by LOCATE, SEQUENTIAL is assumed.

ENTRY Attribute

The ENTRY attribute specifies that the identifier being declared is an entry name. It also is used to describe the attributes of parameters of the entry point.

General format:

```
ENTRY [(parameter-attribute-list
      [,parameter-attribute-list]...)]
```

Each "parameter attribute list" describes the attributes of a single parameter; the parameter name is not listed, but if the parameter is a structure, the level number must precede the attributes for each level. If a parameter is an array, the dimension attribute must be the first specified for that parameter; otherwise, attributes may appear in any order. Parameter attribute lists must appear in the same order as the associated parameters. If the attribute of any parameter need not be described, the absence of the corresponding parameter attribute list must be indicated by a comma.

General rules:

1. The ENTRY attribute with associated parameter attribute lists must be declared for any entry name that is invoked within the block if the attributes of any argument of the invocation differ from the attributes of the associated parameter. This specifies that the compiler is to create the necessary dummy arguments.

2. The ENTRY attribute, without any parameter attribute list, is implied by the attributes REDUCIBLE, IRREDUCIBLE,

USES, SETS, and RETURNS. The term "entry name" is applied to names that are explicitly declared with the ENTRY attribute, to names that receive the ENTRY attribute contextually or by implication, and to names with the BUILTIN or GENERIC attribute.

3. The ENTRY attribute cannot be specified with the BUILTIN or GENERIC attribute.

4. The ENTRY attribute must be specified or implied for an entry name that is a parameter.

5. Expressions used for length, sizes, or bounds in an ENTRY attribute specification for non-controlled parameters are evaluated upon entry to the block to which the declaration of the ENTRY attribute is internal. Such evaluated ENTRY attributes form part of the environment of those blocks internal to the block containing the ENTRY attribute specifications, which are dynamic descendants of that block.

6. Factoring of attributes is not permitted within parameter attribute lists of an ENTRY attribute specification.

7. The ENTRY attribute must appear for each entry name in a GENERIC attribute specification.

8. The ENTRY attribute can be declared for an internal entry name only within the block to which the name is internal.

Assumptions:

The ENTRY attribute can be assumed either contextually or by implication. The appearance of a name as a label prefix of either a PROCEDURE statement or an ENTRY statement constitutes an explicit declaration of that identifier as an entry name. No defaults are applied for parameter attribute lists unless attributes and/or level numbers are specified. If only a level number and/or the dimension attribute is specified for a parameter attribute list, FLOAT, DECIMAL, CONTROLLED, and REAL are assumed.

ENVIRONMENT (File Description Attribute)

The ENVIRONMENT attribute is an implementation-defined attribute that specifies various file characteristics that are not part of the PL/I language.

General format:

```
ENVIRONMENT (option-list)
```

EVENT (Program Control Data Attribute)

The EVENT attribute specifies that the associated identifier being declared is used as an event name. Event names are used to investigate the current state of tasks or of asynchronous input/output operations. They can also be used as program switches.

General format:

```
EVENT
```

General rules:

1. An identifier may be explicitly declared with the EVENT attribute in a DECLARE statement. It may be contextually declared by its appearance in an EVENT option of a CALL statement, in a WAIT statement, in a DISPLAY statement, or in a record transmission statement.

2. Event names may also have the following attributes:

   Dimension

   Scope (the default is INTERNAL)

   Storage class (the default is AUTOMATIC)

   DEFINED (event names may only be defined on other event names)

3. An event variable has two separate values:

   a. A single bit which reflects the completion value of the variable. '1'B indicates complete, '0'B indicates incomplete.

   b. A fixed binary value of implementation-defined precision which reflects the status value of the variable. A zero value indicates normal.

   The values of the event variable can be separately returned by use of the COMPLETION and STATUS built-in functions.

   Assignment of one event variable to another causes both the completion and status values to be assigned. Conversion between event variables and any other data type is not possible.

4. Event variables may be elements of aggregates. Aggregates containing event variables may take part in assignment, provided that this would not require conversion to or from event data.

5. The values of an inactive event variable can be set by one of the following means:

   a. Use of the COMPLETION pseudo-variable, to set the completion value

   b. Use of the STATUS pseudo-variable, to set the status value

   c. Event variable assignment

   d. By a statement with the EVENT option

6. The values of an active event variable can be set by one of the following means:

   a. By a WAIT statement for an event variable associated with an input/output event

   b. By the termination of a task with which the event variable is associated

   c. By closing a file on which an input/output operation with an event option is in progress

   d. Use of the STATUS pseudo-variable, to set the status value

7. An event variable may be associated with an event, that is, a task or an input/output operation, by means of the EVENT option on a statement. The variable remains associated with the event until the event is completed. During this period the event variable is said to be active. It is an error to associate an active event variable with another event, or to modify the completion value of an active event variable by event variable assignment or by use of the COMPLETION pseudo-variable. For a task, the event is completed when the task is terminated because of a RETURN, END, or EXIT statement; for an input/output event, the event is completed during the execution of the WAIT for the associated event.

8. It is an error to assign to the completion value of an active event variable (including an event variable in an array, structure, or area) by means of an input/output statement.

9. On execution of a CALL statement with the EVENT option the event variable, if inactive, is set to zero status value and to incomplete. The sequence of these two assignments is uninterruptable, and is completed before control passes to the named entry point. On termination of the task initiated by the CALL statement, the event variable is set complete and is no longer active. If the task termination is not due to RETURN or END in the task, then the event variable status is set to 1, unless it is already nonzero. The sequence of the two assignments to the event variable values is uninterruptable.

10. On execution of an input/output statement with the EVENT option, the event variable, if inactive, is set to zero status value and to incomplete. The sequence of these two assignments is uninterruptable and is completed before any transmission is initiated but after any action associated with an implicit opening is completed. An input/output event variable will not be set complete until either the termination of the task that initiated the event or the execution, by that task, of a WAIT statement naming the associated event variable. The WAIT operation delays execution of this task until any transmission associated with the event is terminated. If no input/output conditions are to be raised for the operation, the event variable is set complete and is no longer active. If any input/output conditions are to be raised, the event variable is set to have a status value of 1 and the relevant conditions are raised. On normal return from the last on-unit entered as a result of these conditions, or on abnormal return from one of the on-units, the event variable is set complete and is no longer active.

11. An event variable declared for use as a program switch is never set active. Completion and status values must be set by the programmer.

EXCLUSIVE (File Description Attribute)

The EXCLUSIVE attribute specifies that records in a DIRECT UPDATE file may be locked by an accessing task to prevent other tasks from interfering with an operation.

General format:

EXCLUSIVE

General rules:

1. The EXCLUSIVE attribute can be applied to RECORD KEYED DIRECT UPDATE files only.

2. A READ statement referring to a record in an EXCLUSIVE file has the effect of locking that record, unless the READ statement has the NOLOCK option, or unless the record has already been locked by another task; in the latter case, the task executing the READ statement will wait until the record is unlocked before proceeding.

3. Execution in the locking task of a WRITE, DELETE, or REWRITE statement specifying the key of a locked record will automatically unlock the record at the end of the DELETE, REWRITE, or WRITE operation; if the record has been locked by another task, the task executing the WRITE, DELETE, or REWRITE statement will wait until the record is unlocked. While a WRITE, DELETE, or REWRITE operation is taking place, the record is always locked.

4. Automatic unlocking takes place at the end of the operation, on normal return from any on-units entered because of the operation (that is, at the corresponding WAIT statement when the EVENT option has been specified).

5. A locked record can be explicitly unlocked by the task that locked it, by means of the UNLOCK statement.

6. Closing an EXCLUSIVE file unlocks all the records in the file.

7. When a task is terminated, all records locked by that task are unlocked.

Assumptions:

1. If a file is implicitly opened by the UNLOCK statement, it is given the EXCLUSIVE attribute.

2. EXCLUSIVE implies RECORD, KEYED, DIRECT, and UPDATE.

EXTERNAL and INTERNAL (Scope Attributes)

The EXTERNAL and INTERNAL attributes specify the scope of a name. INTERNAL specifies that the name can be known only in the declaring block and its contained blocks. EXTERNAL specifies that the name may be known in other blocks containing an external declaration of the same name.

General format:

EXTERNAL|INTERNAL

Assumptions:

INTERNAL is assumed for entry names of internal procedures and for variables with any storage class. EXTERNAL is assumed for file names and entry names of external procedures. Programmer-defined condition names are assumed to be EXTERNAL.

FILE (File Description Attribute)

The FILE attribute specifies that the identifier being declared is a file name.

General format:

FILE

Assumptions:

The FILE attribute can be implied by any of the other file description attributes. In addition, an identifier may be contextually declared with the FILE attribute through its appearance in the FILE option of any input/output statement, or in an ON statement for any input/output condition.

FIXED and FLOAT (Arithmetic Data Attributes)

The FIXED and FLOAT attributes specify the scale of the arithmetic variable being declared. FIXED specifies that the variable is to represent fixed-point data items. FLOAT specifies that the variable is to represent floating-point data items.

General format:

FIXED|FLOAT

General rule:

The FIXED and FLOAT attributes cannot be specified with the PICTURE attribute.

Assumptions:

Undeclared identifiers (or identifiers declared only with one or more of the dimension, ABNORMAL, NORMAL, DEFINED, SECONDARY, INITIAL, ALIGNED, UNALIGNED, scope, and storage class attributes) are assumed to be arithmetic variables with assigned attributes depending upon the initial letter. For identifiers beginning with any letter I through N, the default attributes are REAL FIXED BINARY with default precision. For identifiers beginning with any other alphabetic character, the default attributes are REAL FLOAT DECIMAL with default precision. If BINARY or DECIMAL and/or REAL or COMPLEX are specified, FLOAT is assumed; however, if a base or mode attribute is specified with a precision attribute that included a scale factor, FIXED is assumed.

FLOAT (Arithmetic Data Attribute)

See FIXED.

GENERIC (Entry Name Attribute)

The GENERIC attribute is used to define a name as a family of entry names, each of which is referred to by the name being declared. When the generic name is referred to, the proper entry name is selected, based upon the arguments specified for the generic name in the procedure reference.

General format:

```
     GENERIC (entry-name-declaration
             [,entry-name-declaration] ...)
```

General rules:

1. No other attributes can be specified for the name being given the GENERIC attribute.

2. Each "entry name declaration" following the GENERIC attribute corresponds to one member of the family, and has the form:

   entry-name attribute-list

3. The "attribute list" of each entry name declaration specifies attributes of the entry name. It must include the ENTRY attribute. It may optionally have USES, SETS, REDUCIBLE, IRREDUCIBLE, INTERNAL, EXTERNAL, and RETURNS attributes. No entry name declaration can have the GENERIC attribute, nor can it have the BUILTIN attribute.

4. Each entry name declaration must specify attributes and/or level numbers for each parameter. An ENTRY declaration within a GENERIC declaration is exactly the same as any other ENTRY declaration. Therefore, no other entry attribute declaration for the same identifier can appear in the same block if the entry name appears in a GENERIC attribute specification.

5. When a generic name is referred to, the attributes of the arguments must match exactly the list following the entry name declaration of one <u>and only one</u> member of the family. The reference is then interpreted as a reference to that member. Thus, the selection of a particular entry name is based upon the arguments of the reference to the generic name. Note that no conversion is done for arguments passed to generic functions. Consequently, the precision of a constant or any other expression must match the precision of a parameter.

6. The selection of a particular entry name is first based on the number of arguments in the reference to the name. The following attributes are then considered in choice of generic members:

   Base

   Scale

   Mode

   Precision

   PICTURE

   LABEL (but not label list)

   Number of dimensions (but not bounds)

   CHARACTER (but not length)

   BIT (but not length)

   VARYING

   ENTRY (but not parameter description or other attributes of entry names)

   FILE (but no other FILE attributes)

   ALIGNED

   UNALIGNED

   AREA (but not size)

   OFFSET (but not specified area variable)

POINTER

TASK

EVENT

7. Generic entry names (as opposed to references) may be specified as arguments to non-generic procedures if the invoked entry name is explicitly declared with the ENTRY attribute. This ENTRY attribute must specify that the appropriate parameter is an entry name and must specify, by means of a further ENTRY attribute, the attributes of all its parameters. This enables a choice to be made of which family member is to be passed.

INITIAL (Data Attribute)

The INITIAL attribute has two forms. The first specifies an initial constant value to be assigned to a data item when storage is allocated to it. The second form specifies that, through the CALL option, a procedure is to be invoked to perform initialization at allocation.

General format:

1. INITIAL (item [,item]...)

2. INITIAL CALL entry-name
   [argument-list]

General rule:

The INITIAL attribute cannot be given for entry names, file names, defined variables, structures, parameters, cell names, or task or event variables. Note, however, that it can be given for an element of a structure or one alternative of a cell (unless an alternative contains an area, in which case only that alternative can be initialized).

Rules for general format 1:

1. In this discussion, the term "constant" denotes one of the following:

   [+|-] arithmetic-constant

   character-string-constant

   bit-string-constant

   [+|-]real-constant{+|-}imaginary-
     constant

2. Only one constant value can be specified for an element variable; more than one can be specified for an array variable. A structure variable can be initialized only by separate initialization of its elementary names, whether they are element or array variables.

3. Constant values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly).

4. If too many constant values are specified for an array, excess ones are ignored; if not enough are specified, the remainder of the array is not initialized.

5. Each item in the list can be a constant, an asterisk denoting no initialization for a particular element, or an iteration specification.

6. The iteration specification has one of the following general forms:

   (iteration-factor) constant

   (iteration-factor)(item[,item]...)

   (iteration-factor) *

   The "iteration factor" specifies the number of times the constant, item list, or asterisk is to be repeated in the initialization of elements of an array. If a constant follows the iteration factor, then the specified number of elements are to be initialized with that value. If a list of items follows the iteration factor, then the list is to be repeated the specified number of times, with each item initializing an element of the array. If an asterisk follows the iteration factor, then the specified number of elements are to be skipped in the initialization operation.

7. The iteration factor is a scalar expression; for STATIC data, it must be an unsigned decimal integer constant. When storage is allocated for the array, the expression is evaluated to give an integer that specifies the number of iterations.

8. A negative or zero iteration factor causes no initialization.

9. For initialization of a string array, if only one parenthesized element expression precedes the string initial value, the expression is interpreted to be a string repetition factor for the string; that is, it is interpreted as a part of the specification of the value for a _single_ element of the

array. Consequently, to cause initialization of more than one element of a string array, both the string repetition factor and the iteration factor must be explicitly stated, <u>even if the string repetition factor is (1).</u> For example, consider the following:

```
((2) 'A') is equivalent to ('AA')
  (for a single element)

((2)(1)'A') is equivalent to
  ('A', 'A') (for two elements)
```

10. Label constants given as initial values for label variables must be known within the block in which the label variable declarations occur. STATIC label variables cannot have the INITIAL attribute.

11. An alternative method of initialization is available for elements of arrays of non-STATIC statement label variables: an element of a label array can appear as a statement prefix, provided that all subscripts are optionally signed decimal integer constants. The effect of this appearance is the initialization of that array element to a value that is a constructed label constant for the statement prefixed with the subscripted reference. This statement must be internal to the block containing the declaration of the array. Only one form of initialization can be used for a given label array.

12. General format 1 of the INITIAL attribute cannot be used in the declaration of locator or area variables.

Rules for general format 2:

1. The "entry name" and "argument list" passed must satisfy the condition stated for prologues in Chapter 6, "Dynamic Program Structure."

2. General format 2 cannot be used to initialize STATIC data.

3. General format 2 can be used to initialize locator and area variables.

Examples:

```
1.  DECLARE SWITCH BIT (1)
            INITIAL ('1'B);

2.  DECLARE MAXVALUE INITIAL (99),
            MINVALUE INITIAL (-99);

3.  DECLARE A (100,10) INITIAL
            ((920)0, (20) ((3)5,9));

4.  DECLARE TABLE (20,20) INITIAL
            CALL INITIALIZE (X,Y);

5.  DECLARE Z(3) LABEL;
        .
        .
        .
    Z(1): IF X = Y THEN GO TO EXIT;
        .
        .
        .
    Z(2): A = A + B + C * D;
        .
        .
        .
    Z(3): A = A + 10;
        .
        .
        .
    GO TO Z(I);
        .
        .
        .
    EXIT: RETURN;
```

The third example results in the following: each of the first 920 elements of A is set to 0, the next 80 elements consist of 20 repetitions of the sequence 5,5,5,9.

In the fourth example, INITIALIZE is the name of a procedure that sets the initial values of elements in TABLE. X and Y are arguments passed to INITIALIZE.

In the last example, transfer is made to a particular element of the array Z by giving I a value of 1,2, or 3.

INPUT, OUTPUT, and UPDATE (File Description Attributes)

The INPUT, OUTPUT, and UPDATE attributes indicate the function of the file. INPUT specifies that data is to be transmitted from external storage to the program. OUTPUT specifies that a new data set is to be created to which data is to be transmitted from the program to external storage. UPDATE specifies that the data can be transmitted in either direction; that is, the file is both an input and an output file.

General format:

```
INPUT|OUTPUT|UPDATE
```

General rules:

1. A file with the INPUT attribute cannot have the PRINT attribute.

2. A file with the OUTPUT attribute cannot have the BACKWARDS attribute.

3. A file with the UPDATE attribute cannot have the STREAM, BACKWARDS, or PRINT attributes. A declaration of UPDATE for a SEQUENTIAL file indicates the update-in-place mode; to access such a file, the sequence of statements must be READ, then REWRITE.

Assumptions:

Default is INPUT. The PRINT attribute implies OUTPUT. If a file is opened implicitly by a PUT, LOCATE, or WRITE statement, OUTPUT is assumed; by a GET or REAL statement, INPUT is assumed; by a DELETE, UNLOCK, or REWRITE statement, UPDATE is assumed. The EXCLUSIVE attribute implies UPDATE.

INTERNAL (Scope Attribute)

See EXTERNAL.

IRREDUCIBLE and REDUCIBLE (Optimization Attributes)

The IRREDUCIBLE and REDUCIBLE optimization attributes, specified for an entry name, supply information to the compiler concerning the degree of optimization that can be accomplished. The IRREDUCIBLE attribute specifies that a calling sequence must be generated for every reference to the entry name. The REDUCIBLE attribute specifies that references to the entry name with arguments of identical values and attributes can always be assumed to have the same effect.

General format:

IRREDUCIBLE|REDUCIBLE

General Rules:

1. Either external and internal procedures can be irreducible or reducible. Blocks invoking procedures that are reducible must be within the scope of a REDUCIBLE, USES, or SETS declaration for the invoked entry name.

2. An external procedure is irreducible if it or any procedure invoked by it:

   a. Access, allocate, modify, or free external data.

   b. Modify, allocate, or free their arguments.

   c. Return inconsistent function values for the same argument values.

   d. Maintain any kind of history.

   e. Perform input/output operations.

   f. Return control from the procedure by means of a GO TO statement.

3. An internal procedure is irreducible:

   a. Under any of the conditions listed under 2 for external procedures.

   b. If it or any procedures called by it access, modify, allocate, or free variables declared in an outer block.

4. Any procedure to which none of the conditions stated in 2 and 3 apply is said to be reducible, and its entry name should be explicitly declared with the REDUCIBLE attribute. The scope of the explicit declaration must include any invoking block.

5. An entry name for which the USES and/or SETS attributes are specified is REDUCIBLE by default. This specifies that the neighborhood of the call can be optimized although the number of references to the entry can be reduced only if no variable is mentioned in both the USES and the SETS list for the entry and if none of the variables named in the USES and SETS lists has its value changed between references.

Assumptions:

Default is IRREDUCIBLE. If USES and/or SETS is specified, the entry is assumed to be definitively reducible.

KEYED (File Description Attribute)

The KEYED attribute specifies that the options KEY, KEYTO, and KEYFROM may be used to access records in the file. These options indicate that keys are involved in accessing the records in the file.

General format:

```
KEYED
```

General rules:

1. A KEYED file cannot have the attributes STREAM or PRINT.

2. The KEYED attribute can be specified for RECORD files only.

3. The KEYED attribute must be specified for every file with which any of the options KEY, KEYTO, and KEYFROM is used. It need not be specified if none of the options are to be used, even though the corresponding data set may actually contain recorded keys.

Assumption:

The DIRECT and EXCLUSIVE attributes imply KEYED.

LABEL (Program Control Data Attribute)

The LABEL attribute specifies that the identifier being declared is a label variable and is to have statement labels as values. To aid in optimization of the object program, the attribute specification may also include the values that the name can have during execution of the program.

General format:

```
LABEL [(statement-label-constant
      [,statement-label-constant]...)]
```

General rules:

1. If a list of statement label constants is given, the variable can have as values only members of the list. If multiple labels are prefixed to a statement all of the labels have the same value. The label constants in the list must be known in the block containing the declaration.

2. The parenthesized list of statement label constants can be used in a LABEL attribute specification for a label array. The label list applies to each element of the array.

3. If the variable is a parameter, its value can be any statement label variable or constant passed as an argument. If the argument is a label variable, the value of the label parameter can be any value permitted for the label variable that is passed.

4. An entry name cannot be a value of a label variable.

5. A subscripted label specifying an element of a label array can appear as a statement label prefix, if the label variable is not STATIC, but it cannot appear in an END statement after the keyword END. For further information, see general rule 12 in the discussion of the INITIAL attribute.

6. The INITIAL attribute cannot be specified for STATIC label variables.

Length (String Attribute)

See BIT.

LIKE (Structure Attribute)

The LIKE attribute specifies that the name being declared is a structure variable with the same structuring as that for the name following the attribute keyword LIKE. Substructure names, elementary names, and attributes for substructure names and elementary names are to be identical.

General format:

```
LIKE structure-variable
```

General rules:

1. The "structure variable" can be a major structure name or a minor structure name. It can be a qualified name, but it cannot be subscripted.

2. The "structure variable" must be known in the block containing the LIKE attribute specification. The structure names in all LIKE attributes are associated with declared structures before any LIKE attributes are expanded. For example:

```
DECLARE 1 A, 2 C, 3 E, 3 F,
        1 D, 2 C, 3 G, 3 H;
   .
   .
   .
BEGIN;
  DECLARE 1 A LIKE D, 1 B LIKE A.C;
   .
   .
   .
END;
```

These declarations result in the following:

```
1 A LIKE D is expanded to give:

  1 A, 2 C, 3 G, 3 H

1 B LIKE A.C is expanded to give:

  1 B, 3 E, 3 F
```

3. Neither the "structure variable" nor any of its substructures can be declared with the LIKE attribute, nor may the "structure variable" have been completed by the LIKE attribute.

4. Neither additional substructures nor elementary names can be added to the created structure; any level number that immediately follows the "structure variable" in the LIKE attribute specification in a DECLARE statement must be algebraically equal to or less than the level number of the name declared with the LIKE attribute.

5. Attributes of the "structure variable" itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the "structure variable" following the keyword LIKE represents an array of structures, its dimension attribute is not carried over. The only ALIGNED and UNALIGNED attributes that are carried over are those explicitly specified for substructures and elements of the structure variable; the LIKE attribute is expanded before the ALIGNED and UNALIGNED attributes are applied to the contained elements of the "structure variable." The other attributes of substructure names and elementary names, however, are carried over; if the attributes that are carried over contain names, these names are interpreted in the block containing the LIKE attribute. An exception is that this does not apply to the INITIAL attribute for any elements of a label array that has been initialized by prefixing to a statement.

6. If a direct application of the description to the structure declared LIKE would cause an incorrect continuity of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1) the level numbers are modified by a constant before application.

NORMAL (Optimization Attribute)

See ABNORMAL.

OFFSET and POINTER (Program Control Data Attributes)

The OFFSET and POINTER attributes describe locator variables. A locator variable can be used in a based variable reference to identify a particular generation of the based variable. Offset variables identify a location relative to the start of an area; pointer variables identify any location, including those within areas.

General format:

```
POINTER|OFFSET[(scalar-area-variable)]
```

General rules:

1. A pointer variable can be explicitly declared in a DECLARE statement, or it can be contextually declared by its appearance as a pointer qualifier, by its appearance in a BASED attribute, or by its appearance in a SET option.

2. An offset variable must be explicitly declared.

3. The value of a pointer variable or function uniquely identifies a generation. This generation may be accessed by using the variable or function as the locator qualifier in a reference to a based variable whose evaluated attributes match those of the generation. A value of pointer type may be obtained from the built-in functions ADDR, NULL, and POINTER.

4. The value of an offset variable or function identifies the position of a generation within an area relative to the area. This value may be converted to a pointer to the generation by supplying the area and the offset value as arguments to the POINTER built-in function. A value of offset type may be obtained from the built-in functions NULLO and OFFSET. If an offset, O, when associated (e.g., by the POINTER built-in function) with an area A1, identifies a generation G1, then when A1 is assigned to A2 (possibly by some intervening input/output operations) the generation G2 in A2 which corresponds to G1 may be accessed by the pointer value obtained by supplying A2 and O to the POINTER built-in function. Use of an offset to access a generation in an area other than the area initially used to establish the offset is allowed in more cases than the foregoing. The general case is now given, using the foregoing nomenclature. There can be associated with an area an ordered list of the evaluated

attributes of the significant allocations (see "The AREA Attribute") made in the area. G2 is accessed by POINTER (A2,O) provided the ordered list of evaluated attributes of A1 when G1 was allocated match the part, up to the allocation of G2, of the list of evaluated attributes of the significant allocations in A2 when O is used to access G2.

5. The value of a locator variable can be set in any of the following ways:

   a. With the SET option of a READ statement

   b. By a LOCATE statement

   c. By an ALLOCATE statement

   d. By assignment of the value of a locator variable or function

6. Locator variables cannot be operands of any operators other than the comparison operators = and ¬=.

7. Locator data cannot be converted to any other data type, but pointer can be converted to offset, and vice versa.

8. A locator value can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored.

9. Locator data cannot be transmitted using STREAM input/output.

10. Only the INITIAL CALL form of the INITIAL attribute is allowed in locator declarations.

Assumptions:

An undeclared identifier appearing in the BASED attribute specification, in a SET option, or as a locator qualifier, is contextually declared to be a pointer variable. An undeclared identifier appearing in the OFFSET attribute specification is contextually declared to be an area variable. A variable named in the OFFSET attribute is given the AREA attribute.

OUTPUT (File Description Attribute)

See INPUT.

PICTURE (Data Attribute)

The PICTURE attribute is used to define the internal and external formats of character-string, numeric character, and numeric bit data and to specify the editing of data. Numeric character data is data having an arithmetic value but stored internally in character form. Numeric character data is converted to coded arithmetic before arithmetic operations can be performed.

The picture characters are described in Appendix 2, "Picture Specification Characters."

General format:

```
PICTURE

    'character-picture-specification'

    'numeric-picture-specification'
```

A "picture specification," either character or numeric, is composed of a string of picture characters enclosed in single quotation marks. An individual picture character may be preceded by a repetition factor, which is a decimal integer constant, _n_, enclosed in parentheses, to indicate repetition of the character _n_ times. If _n_ is zero, the character is ignored. Picture characters are considered to be grouped into _fields_, some of which contain _subfields_.

General rules:

1. The "character picture specification" is used to describe a character-string data item. Three characters may be used: A, indicating that the associated position in the data item may contain any alphabetic character or a blank; X, indicating that the associated postion may contain any character; and 9, indicating that the associated position may contain any decimal digit or a blank. A character picture specification must include at least one A or X. Each character picture specification is a single field with no contained subfields.

   Example:

   ```
   DECLARE ORDER# PICTURE
           'AA(3)9X99X(4)9';
   ```

   This declaration specifies that values of ORDER# are to be character strings of length 13. The string consists of two letters, three digits, any character, two digits, any character, and four digits. For example, the character string 'GF342-63-0024' would fit this description.

Editing and suppression characters are not allowed in character picture specifications. Each picture specification character must represent an actual character in the data item.

The "numeric picture specification" is used to describe, for decimal digits, a character item that represents either an arithmetic value or a character-string value, depending upon its use. For binary digits, the "numeric picture specification" is used to describe a bit item that represents either an arithmetic value or a bit-string value. A numeric picture specification can consist of one or more fields, some of which can be divided into subfields. A single field is used to describe a fixed-point number or the mantissa of a floating-point number. Either may be divided into two subfields, one describing the integer portion, the other describing the fractional portion. For floating-point numbers, a second field is required to describe the exponent; it cannot be divided into subfields. A second field may optionally be used with fixed-point numbers to indicate a scaling factor. Seven basic picture characters can be used in a numeric picture specification:

9 indicating any decimal digit

1 indicating any binary digit

2 indicating a binary digit in 2's complement notation

3 indicating a binary digit in 1's complement notation

V indicating the assumed location of a decimal point. It does not specify an actual character in the character-string value of the data item. The V also indicates the end of a subfield of a picture specification.

K indicating, for floating-point data items, that the exponent should be assumed to begin at the position associated with the picture character following the K. It does not specify an actual character in the character-string value of the data item, either an E or a sign. The K delimits the two fields of the specification.

E indicating, for floating-point data items, that the associated position will contain the letter E to indicate the beginning of the exponent. The E also delimits the two fields.

In addition to these characters, zero suppression characters, editing characters, and sign characters may be included in a numeric picture specification to indicate editing. Editing characters are not a part of the arithmetic value of a numeric character data item, but they are a part of its character-string value. Repetition factors are allowed in numeric picture specifications.

3. A numeric character data item can have a decimal or binary base, depending upon the digit picture character used. Its scale and precision are specified by the picture characters. The PICTURE attribute cannot be specified in combination with base, scale, or precision attributes. If the mode of the numeric character data is COMPLEX, however, the COMPLEX attribute must be explicitly stated.

4. The following paragraphs indicate the combinations of picture characters for different arithmetic data formats.

   a. Real decimal fixed-point items are described in the following general form:

```
PICTURE '[9]...[V][9]...
        [F([+|-] integer)]'
```

   The optional field of the picture specification, beginning with the letter F together with a parenthesized, optionally signed decimal integer constant, is a scaling factor that indicates the location of an assumed decimal point if that location is outside the actual data item. The scaling factor has an effect similar to the exponent of a floating-point number; it indicates that the assumed decimal point is "integer" places to the right (or left, if negative) of the position otherwise indicated.

   Sign, editing, and zero suppression picture characters can be included in a fixed-point specification. The V cannot appear more than once in a specification, although it may be used in combination with the decimal point (.) or comma (,) editing characters, which cause insertion of a period or comma. If no V is included, the decimal point is assumed to be to the right of the rightmost digit. Only one sign indication can be included in the first field (the actual sign of the integer in a scaling factor is allowed additionally). The specification must include at least one digit position.

Example:

```
DECLARE A PICTURE '999V99';
```

This specification describes numeric character items of five digits, two of which are assumed to be fractional digits.

b. Real decimal floating-point items are described by the following general form:

```
PICTURE
   '[9]...[V][9]...{E|K}9...'
```

Both the mantissa field and the exponent field must each contain at least one digit position.

Sign, editing, and zero suppression picture characters can be included in a floating-point specification. One sign indication is allowed for each field. Only one V is allowed, and it can appear in the first field only. As with fixed-point specifications, the V may appear in combination with the decimal point editing character (as .V or V.). At least one digit must appear in the mantissa field.

c. Real binary fixed-point items are described in the following general forms:

```
PICTURE '[S][1]...[V][1]...
         [F([+|-]integer)]'

PICTURE '[2]...[V][2]...
         [F([+|-]integer)]'

PICTURE '[3]...[V][3]...
         [F([+|-]integer)]'
```

Note: The picture character 1 specifies that the associated position in the data contains a binary digit. The picture character 2 specifies that the associated position in the data contains a binary digit that is a part of a binary value in 2's complement notation. The picture character 3 specifies that the associated position in the data contains a binary digit that is a part of a binary value in 1's complement notation. A binary picture specification cannot contain a combination of the characters 1, 2, and 3.

Only one V, representing a point, can be present in a picture specification, but it may be in any position within the first (or only) field. When a sign character (S) is specified, the data will contain a binary 1, if the value is negative, or a zero, if the value is positive. The sign character can be used only with the picture character 1. At least one digit must appear in the mantissa field.

No picture characters other than those shown above can be used in a real binary fixed-point picture specification.

d. Real binary floating-point items are described in the following general forms:

```
PICTURE '[S][1]...[V][1]...
         K[S]1[1]...'

PICTURE '[2]...[V][2]...K2[2]...'

PICTURE '[3]...[V][3]...K3[3]...'
```

(See the note in paragraph c, above, for an explanation of the picture characters 1, 2, and 3.) At least one digit must appear in the mantissa field.

The sign character allowed to the right of the K when the picture character 1 is used represents the sign of the exponent. Signs are not allowed with specifications using either the picture character 2 or the picture character 3.

Note that the exponent is expressed in binary notation and that the picture character E is not allowed in the picture specification nor is an actual E allowed to appear in the data.

No characters other than those shown in the format above can be used in a binary floating-point picture specification.

e. Complex numeric character data is described using the general form:

```
PICTURE 'real-picture' COMPLEX
```

The "real picture" is a specification for either a fixed-point or a floating-point data item. The single picture specification describes both parts of a complex number.

5. The precision of a numeric character variable is dependent upon the number of digit positions, actual and conditional. Digit positions can be specified by the following characters:

   9 which is an actual decimal digit character

   1 which is an actual binary digit character

   2 which is an actual binary digit character for a 2's complement number

   3 which is an actual binary digit character for a 1's complement number

   Z, *, Y which are conditional decimal digit characters specifying zero suppression

   T, I, R which are decimal digit characters specifying an overpunch

   $, +, -, S which are conditional decimal digit drifting characters

   Each but the first conditional digit drifting character in a drifting string specifies a digit position. A conditional digit drifting character used alone does not specify a digit position.

   Precision of a fixed-point variable is (p,q), where p is the number of digit positions in the picture specification and q is the number of digit positions following V. Precision of a floating-point variable is (p), where p is the number of digit positions preceding the E or K. Indicated static editing characters or insertion characters do not participate in the specification of precision, but they must be counted in the number of characters if the data item is written as output or assigned internally to a character string.

6. A variable representing sterling data items can be specified by using a numeric picture specification that consists of three fields, one each for pounds, shillings, and pence. The pence field may be divided into two subfields. Data so described is stored in character format as three contiguous numbers corresponding to each of the three fields. If any arithmetic operations are specified for the variable, its value is converted to coded fixed-point decimal representing the value in pence. Sterling picture specifications have the following form:

```
PICTURE

   'G [editing-character-1]...

   M pounds-field

   M [separator-1]...
         shillings-field

   M  [separator-2]...
         pence-field

         [editing-character-2]...'
```

   Picture specification characters, editing characters, and separators can be used in any of these fields and are discussed in Appendix 2, "Picture Specification Characters."

   The precision (p,q) of a sterling numeric character data item is defined as follows:

   q = number of fractional digits in the pence field

   p = 3+q+(number of digit positions, actual and conditional, in the pounds field)

POINTER (Program Control Data Attribute)

See OFFSET.

POSITION (Data Attribute)

See DEFINED.

Precision (Arithmetic Data Attribute)

The precision attribute is used to specify the minimum number of significant digits to be maintained for the values of the data items, and to specify the scale factor (the assumed position of the binary or decimal point). The precision attribute applies to both binary and decimal data.

General format:

    (number-of-digits [,scale-factor])

The "number of digits" is an unsigned decimal integer constant and "scale factor" is an optionally signed decimal integer constant. The precision attribute specification is often represented, for brevity, as (p,q), where p represents the "number of digits" and q represents the "scale factor."

General rules:

1. The precision attribute, if it appears, must immediately follow the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) attribute at the same factoring level.

2. The number of digits specifies the number of digits to be maintained for data items assigned to the variable. The scale factor specifies the number of fractional digits. No point is actually present; its location is assumed.

3. The scale factor can be specified for fixed-point variables only; the number of digits can be specified for both fixed-point and floating-point variables.

4. When the scale is FIXED and no scale factor is specified, it is assumed to be zero; that is, the variable is to represent integers.

5. The scale factor can be negative, and it can be larger than the number of digits. A negative scale factor (-q) always specifies integers, with the point assumed to be located q places to the right of the rightmost actual digit. A positive scale factor (q) that is larger than the number of digits always specifies a fraction, with the point assumed to be located q places to the left of the rightmost actual digit.

6. The precision attribute cannot be specified in combination with the PICTURE attribute.

Assumptions:

The defaults are implementation defined and dependent upon the base and scale of the variable.

PRINT (File Description Attribute)

The PRINT attribute specifies that the data of the file is ultimately to be printed. The PAGE and LINE options of the PUT statement and the PAGESIZE option of the OPEN statement can be used only with files having the PRINT attribute.

General format:

    PRINT

General rules:

1. The PRINT attribute implies the OUTPUT and STREAM attributes.

2. The PRINT attribute conflicts with the RECORD attribute.

REAL (Arithmetic Data Attribute)

See COMPLEX.

RECORD and STREAM (File Description Attributes)

The RECORD and STREAM attributes specify the kind of data transmission to be used for the file. STREAM indicates that the data of the file is considered to be a continuous stream of data items, in character form, to be assigned from the stream to variables, or from expressions into the stream. RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable or buffer.

General format:

    RECORD|STREAM

General rules:

1. A file with the STREAM attribute can be specified only in the OPEN, CLOSE, GET, and PUT I/O statements.

2. A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, UNLOCK, and DELETE I/O statements.

3. A file with the STREAM attribute cannot have any of the following attributes: UPDATE, DIRECT, SEQUENTIAL, BACKWARDS, BUFFERED, UNBUFFERED, EXCLUSIVE, and KEYED, any of which implies RECORD.

4. A file with the RECORD attribute cannot have the PRINT attribute.

Assumptions:

Default is STREAM. If a file is implicitly opened by a READ, WRITE, REWRITE, LOCATE, UNLOCK, or DELETE statement, RECORD is assumed.

REDUCIBLE (Optimization Attribute)

See IRREDUCIBLE.

RETURNS (Entry Name Attribute)

The RETURNS attribute may be specified in a DECLARE statement for an entry name that is used in a function reference within the scope of the declaration. It is used to describe the attributes of the function value returned when that entry name is invoked as a function.

General format:

```
RETURNS (attribute...)
```

General rules:

1. The attributes in the parenthesized list following the keyword RETURNS must be separated by blanks (except for attributes such as precision that are enclosed in parentheses). They must agree with the attributes specified explicitly or by default in the PROCEDURE or ENTRY statement to which the entry name is prefixed.

2. Only arithmetic, string, locator, AREA, PICTURE, ALIGNED, and UNALIGNED attributes can be specified.

3. Length attribute specifications are evaluated on entry to the block containing the RETURNS attribute specification. Such evaluated RETURNS attributes form part of the environment of blocks contained within the block declaring the attribute and dynamically descendant from the block.

4. For an internal function, the RETURNS attribute can be specified only in a DECLARE statement that is internal to the same block as the function procedure.

Assumptions:

If the RETURNS attribute is not specified for an entry name, a RETURNS attribute is assumed specifying the attributes REAL, FIXED, BINARY with default precision if the entry name begins with any of the letters I through N; otherwise, the assumed attributes are REAL, FLOAT, DECIMAL with default precision.

SECONDARY Attribute

Function:

The SECONDARY attribute is used to specify that certain data normally does not require efficient storage.

General format:

```
SECONDARY
```

General rules:

1. This attribute may be declared only for major structures, arrays, and variables not contained in structures or arrays, i.e., for variables at level 1.

2. The attribute specifies that where possible and necessary, less than normally efficient storage may be allocated to the variable.

SEQUENTIAL (File Description Attribute)

See DIRECT.

SETS and USES (Optimization Attributes)

The SETS and USES attributes specify, for an entry name, the nature of its irreducibility due to data manipulation. The SETS attribute specifies all of the data, including arguments, that may be altered, allocated, or freed by the procedure, or any procedures called by it. The USES attribute specifies all of the data (though not the arguments) that is accessed by the procedure, or any procedures called by it.

General format:

```
USES (item [,item]...)

SETS (item [,item]...)
```

General rules:

1. The items of the list following a USES or SETS attribute can be as follows:

   a. A decimal integer constant indicating the parameter position that is used or set. Thus, a 1 indicates the first parameter, a 2 the second parameter and so on, with the nth parameter being specified by an integer constant of value n.

   b. An unsubscripted, non-based data variable known in both the block containing the declaration and in the invoked procedure. An asterisk can be used as an abbreviated notation to describe all such variables.

2. When an item appears in a USES list, it indicates that the invoked procedure or procedures invoked by it:

   a. Access that item

   b. Do not assign to that item unless it is also specified in a SETS attribute

   c. Do not access any other data known to the block, except data designated by explicit arguments in either a CALL statement or a function reference.

3. When an item is specified in a SETS list, it indicates that the invoked procedure or procedures invoked by it:

   a. Assign to, allocate, or free that item

   b. Do not access that item other than to reassign, allocate, or free it, unless it is also specified in a USES attribute, or it is an argument

   c. Do not assign to, allocate, or free any other data known in the block

4. Items appearing in USES or SETS lists indicate the following:

   a. It is assumed that any item not in a SETS list, but known both inside and outside the procedure, will not be altered by the invocation of the procedure. It is also assumed that any item known both inside and outside the procedure, but not in a USES list, will not be used.

   b. It is assumed that arguments will be used but not set, unless they are in a SETS list.

   c. If a data item represented by a variable known outside the procedure is both used and set within the procedure, it must appear in both the USES and SETS lists.

5. The USES and SETS attributes may be declared for any entry name used to invoke a procedure. The scope of this declaration must include the invoking block. If the ENTRY attribute is not declared, ENTRY is implied. If either USES or SETS is declared in the invoking procedure, complete information must be given about the data that is used and/or set by the invoked procedure.

6. If an item in a USES or SETS list, as described in 1b above, is defined on a base, and if the base and any other items defined on it are known both to the invoking and invoked blocks, the base and the other items must also be specified in the list.

7. A structure or array variable appearing in a USES or SETS list implies that names of all items contained in the structure or array also are in the list. However, it does not imply that items defined on elements of the structure are in the list; these must be declared as in rule 6, above.

8. If a procedure is declared with the USES or SETS attribute, or both, and is not declared to be IRREDUCIBLE, then it is assumed that the procedure is not irreducible for any other reason. If it is (for example, if it performs input/output), then the IRREDUCIBLE attribute must also be specified.

STATIC (Storage Class Attribute)

See AUTOMATIC.

STREAM (File Description Attribute)

See RECORD.

TASK (Program Control Data Attribute)

The TASK attribute describes a variable that may be used as a task name, to test or control the relative priority of a task.

General format:

TASK

General rules:

1. An identifier can be explicitly declared with the TASK attribute in a DECLARE statement, or it can be contextually declared by its appearance in a TASK option of a CALL statement.

2. Task variables can also have the following attributes:

   a. Dimension

   b. Scope (the default is INTERNAL)

   c. Storage class (the default is AUTOMATIC)

   d. DEFINED (task variables may only be defined on other task names)

   e. SECONDARY

3. A task variable can be used in the following contexts:

   a. In the TASK option of a CALL statement

   b. As an argument of the ALLOCATION built-in function and of the PRIORITY pseudo-variable or built-in function.

   c. As an argument in a procedure.

   d. As a parameter in a PROCEDURE or ENTRY statement or in the parameter attribute list of an ENTRY attribute

   e. In an ALLOCATE or FREE statement

4. A task variable may be associated with a task by specifying the task name in the TASK option of a CALL statement. A task variable is said to be active if its associated task is active. A task variable must be in an allocated state when it is associated with a task and must not be freed while it is active. An active task variable cannot be associated with another task.

5. A task variable contains a single value, a priority value. This value is a fixed-point binary value of precision (n,0), where _n_ is implementation-defined. This value can be tested and adjusted by means of the PRIORITY built-in function and pseudo-variable. The built-in function returns the priority of the task argument relative to the priority of the task executing the function. Similarly, the pseudo-variable permits assignment, to the named task variable, of a priority relative to the priority of the task executing the assignment.

6. Structures, arrays, or areas containing task variables cannot take part in assignment or input/output operations.

7. Task data cannot be converted to any other data type.

8. A task variable cannot be passed as an argument if this would require creation of a dummy argument.

UNALIGNED (Data Attribute)

See ALIGNED.

UNBUFFERED (File Description Attribute)

See BUFFERED.

UPDATE (File Description Attribute)

See INPUT.

USES (Optimization Attribute)

See SETS.

VARYING (String Attribute)

See BIT.

PARAMETERS

The PROCEDURE statement heading a given procedure and defining the primary entry point to the procedure may specify a list of parameters.

One or more ENTRY statements may also be used in the procedure to define secondary entry points. Like the heading statement of the procedure, each of the ENTRY statements must have at least one label to serve as an entry name for that point, and each may specify a list of parameters. Parameter lists for different entry points to a procedure need not be the same.

A parameter may be a scalar, array, or structure name that is unqualified and unsubscripted, or it may be a file parameter or an entry parameter. Parameters must be level 1 identifiers, i.e., they cannot be members of structures.

A file parameter may be used within a procedure wherever a file name may be used; an entry parameter may be used wherever an entry name may be used.

A reference within a procedure to a parameter produces an undefined result if the entry point at which the procedure is invoked does not include that parameter in its parameter list.

Parameters are explicitly declared by their appearance in a PROCEDURE or ENTRY statement, but attributes can be supplied in a DECLARE statement internal to the procedure. If attributes are not supplied in a DECLARE statement, default arithmetic attributes are applied, depending upon the initial letter of the identifier.

Parameters cannot be declared with the storage class attributes STATIC, AUTOMATIC, or BASED or with the DEFINED attribute, but a parameter may be used as a base identifier in a DEFINED attribute for simple and iSUB defining.

A parameter may have the CONTROLLED storage class attribute. In this case, the associated argument must also have the CONTROLLED attribute with no dummy created for that argument.

Scope attributes cannot be declared for parameters; internal is always assumed. Except for controlled parameters, any bounds, lengths, and area sizes must be specified either by asterisks or decimal integer constants which, for bounds, may be signed.

Example:

```
 SBPRIM:    PROCEDURE (X, Y, Z);
            DECLARE (X, Y, A, B) FIXED, Z
             FLOAT;
            A = X-1; B = Y+1;
            GO TO COMMON;
 SBSEC:     ENTRY (X, Z);
            A = X-2; B = X-3;
            COMMON:  Z = A**2+A*B+B**2;
            END SBPRIM;
```

In this example, the procedure may be entered at its primary entry point SBPRIM, where the parameter list is (X, Y, Z), or at its secondary entry point SBSEC, where the parameter list is (X, Z).

PROCEDURE REFERENCES

At any point in a program where an entry name for a given procedure is known, the procedure may be invoked by a procedure reference, which has the form:

entry-name [(argument [,argument] ...)]

The number of arguments (possibly zero) in the procedure reference must be equal to the number of parameters in the list for the entry point denoted by the entry name.

The procedure invoked by the procedure reference may be an external or an internal procedure. If it is an internal procedure, the block to which the entry name is internal must be active at the time of invocation of the procedure.

When a procedure reference invokes a procedure, each argument specified in the reference is associated with its corresponding parameter in the list for the denoted entry point, and control is passed to the procedure at the entry point.

When a procedure becomes inactive, the association between arguments and parameters is terminated.

There are two distinctly different uses for procedures, determined by one of two contexts in which a procedure reference may appear:

1. A procedure reference may appear as an operand in an expression. In this case, the reference is said to be a function reference, and the procedure is invoked as a function procedure, or simply a function.

2. A procedure reference may appear following the keyword CALL, either in a CALL statement or in a CALL option. In this case, the reference is said to be a subroutine reference, and the procedure is invoked as a subroutine procedure, or simply a subroutine.

FUNCTION REFERENCES AND PROCEDURES

When a function reference appears in an expression, the procedure is invoked. The procedure is then executed, using the arguments, if any, specified in the function reference. The result of this execution is the required value, which is passed with return of control back to the point of invocation. This returned value is then used to evaluate the expression.

The procedure invoked by a function reference normally will terminate execution with a statement of the form RETURN (expression), where expression is a scalar expression of arithmetic, character-string, bit-string, locator, or area type. It is the value of this expression that will be returned as the function value. The PROCEDURE or ENTRY statement at the invoked entry point may specify data attributes for the function value. Just prior to return, the expression is evaluated, and, before being passed back, the value is converted, if necessary, to conform to these attributes, or, if the attributes are not specified, to the default attributes implied by the entry name.

GENERIC ENTRY NAMES

A generic entry name designates a family of entry points with a single name. A reference to a generic name causes the selection of a certain member of the family, depending upon the attributes of the arguments. The characteristics of the value returned depend upon the member that is selected.

Generic names may be built-in (see below) or specified by the programmer, who may, by means of the GENERIC attribute, define a name to be a generic procedure name. The GENERIC attribute requires a list of all of the entry names of the family and the attributes of all of the parameters for each member (different members must have different parameter attribute lists). Then any reference appearing in the scope of this declaration and using the declared generic name as an entry name will result in the use of that member of the declared family with the parameter attribute lists that match the arguments in the reference.

BUILT-IN FUNCTIONS

Besides function references to procedures written by the programmer, a function reference may invoke one of a comprehensive set of built-in functions.

The set of built-in functions is an intrinsic part of PL/I. It includes not only the commonly used arithmetic functions but also functions for manipulating strings and arrays, as well as other necessary or useful functions related to special facilities provided in the language. The identifiers corresponding to the built-in function names are not reserved; any such identifier can be used by the programmer for other purposes. The complete list of these functions and their descriptions can be found in Appendix 1.

Each built-in function, whether or not it is generic, requires a specified number of arguments. For some built-in functions only a minimum is specified; additional arguments are optional. For others, a maximum is specified.

Each of the built-in functions that is not generic has only a single member. When a reference is made to one of these functions, any arguments whose attributes do not match the attributes required by that function are converted to the appropriate form before the function is invoked. The characteristics of the value returned are determined by the function.

Unlike programmer-specified functions, which always return a scalar value, there are many built-in functions that may return an array or structure value when array or structure expressions are used in certain of their argument positions. This facility is useful in array or structure expressions.

SUBROUTINE REFERENCES AND PROCEDURES

When a procedure is invoked by the execution of a CALL statement or a CALL option, the initial action is the same as if the procedure were invoked as a function: the arguments in the procedure reference, if any, are associated with the parameters, and control is passed to the procedure at the denoted entry point. No value is returned by a procedure invoked in this way.

A procedure may be terminated in one of the following ways:

1. Control reaches a RETURN statement for the procedure. When executed, this statement normally returns control to the first executable statement logically following the invoking statement.

2. Control reaches an END statement for the procedure. The effect is as in case 1.

3. Control reaches a GO TO statement in the procedure that transfers control out of the procedure. In this case, control will go to the designated statement.

4. Control reaches an EXIT or STOP statement.

THE ARGUMENTS IN A PROCEDURE REFERENCE

When a procedure is invoked, a relationship is established between the arguments of the invoking statement and the parameters of the invoked entry point. A parameter itself may be passed as an argument.

In general, the arguments in a procedure reference may be any of the following:

1. Expressions

2. Entry names (programmer-defined)

3. Mathematical built-in function names (see Appendix 1)

4. Filenames

The attributes of each argument in a procedure reference must, in general, match the attributes of the corresponding parameter at the named entry point.

For example, assume that the procedure SUB in a program is defined by:

```
SUB:  PROCEDURE (X, Y, Z);
      DECLARE X FIXED, Y ENTRY, Z LABEL;
      .
      .
      .
      END SUB;
```

This implies that the parameter X is used as a fixed-point variable with certain default data attributes, Y is used as an entry name, and Z is a statement label variable in the body of the procedure. Then if SUB is invoked in the program by the statement:

```
     CALL SUB (R*S, CALC, L5);
```

it is then necessary that:

1. The expression R*S has all the data attributes of the parameter X (unless SUB is described by an ENTRY attribute; see below).

2. CALC be an entry name.

3. L5 be a statement-label designator.

EVALUATION OF ARGUMENT SUBSCRIPTS

When an argument is a subscripted variable, the subscripts are evaluated before invocation. The specified element is then passed as the argument. Subsequent changes in the subscript during the execution of the invoked procedure have no effect upon the corresponding parameter.

USE OF DUMMY ARGUMENTS

A constructed dummy argument containing the argument value is passed to a procedure if the argument is one of the following:

- an arithmetic, string, or label constant
- an expression involving operators
- an expression in parentheses
- an expression whose data attributes disagree with the data attributes declared for the parameter in an ENTRY attribute specification in the invoking block
- a function reference with arguments

In all other cases the argument as it appears is passed. The parameter becomes identical with the passed argument. If a dummy is created, changes to the parameter are not reflected back in the original argument.

ENTRY NAMES AS ARGUMENTS

When an entry name is specified as an argument of a procedure, one of the following applies:

1. If the entry name argument, call it M, is specified with an argument list of its own, it is recognized as a function reference; M is invoked, and the value returned by M effectively replaces M and its argument list in the containing argument list.

2. If the entry name argument appears without an argument list, but within an operational expression or within parentheses, then it is taken to be a function reference with no arguments. For example:

   ```
   CALL A((B));
   ```

   This passes, as the argument to procedure A, the value returned by the function procedure B.

3. If the entry name argument appears without an argument list and neither within an operational expression nor within parentheses, the entry name itself is passed to the function or subroutine being invoked. In such cases, the entry name is not taken to be a function reference, even if it is the name of a function that does not require arguments. For example:

   ```
   CALL A(B);
   ```

   This passes the entry name B as an argument to procedure A.

   There is an exception to this rule, however: if an identifier is known as an entry name and appears as an argument and if the parameter attribute list for that argument specifies an attribute other than an entry name, the entry name will be invoked and its returned value passed. For example:

   ```
   A: PROCEDURE;
      DECLARE B ENTRY,
              C ENTRY(FLOAT);
        .
        .
        .
      X = C(B);
        .
        .
        .
      END A;
   ```

   In this case, B is invoked and its returned value is passed to C.

Consider the following example:

```
CALLP: PROCEDURE;
       DECLARE RREAD ENTRY,
               SUBR ENTRY (ENTRY, FLOAT,
                  FIXED BINARY, LABEL);
         .
         .
         .
       GET LIST (R,S);
         .
         .
         .
       CALL SUBR (RREAD, SQRT(R), S,
          LAB1);
         .
         .
         .
 LAB1: CALL ERRT(S);
         .
         .
         .
       END CALLP;

 SUBR: PROCEDURE(NAME, X, J, TRANPT);
       DECLARE NAME ENTRY, TRANPT LABEL;
         .
         .
         .
       IF X > J THEN CALL NAME(J);
                ELSE GO TO TRANPT;
         .
         .
         .
       END SUBR;
```

In this example, assume that CALLP, SUBR, and RREAD are external entry names. In CALLP, both RREAD and SUBR are explicitly declared to have the ENTRY attribute. (Actually, the explicit declaration for SUBR is used principally to provide information about the characteristics of the parameters of SUBR.) Four arguments are specified in the CALL SUBR statement. These arguments are interpreted as follows:

1. The first argument, RREAD, is recognized as an entry name (because of the ENTRY attribute declaration). This argument is not in conflict with the first parameter as specified in the parameter attribute list in the ENTRY attribute declaration for SUBR in CALLP. Therefore, since RREAD is recognized as an entry name and not as a function reference, the entry name is passed at invocation.

2. The second argument, SQRT(R), is recognized as a function reference because of the argument list accompanying the entry name. SQRT is invoked, and the value returned by SQRT is assigned to a dummy argument, which effectively replaces the reference to SQRT. The attributes of the dummy argument agree with those of the

second parameter, as specified in the parameter attribute list declaration. When SUBR is invoked, the dummy argument is passed to it.

3. The third argument, S, is simply a decimal floating-point element variable. However, since its attributes do not agree with those of the third parameter, as specified in the parameter attribute list declaration, a dummy argument is created containing the value of S converted to the attributes of the third parameter. When SUBR is invoked, the dummy argument is passed.

4. The fourth argument, LAB1, is a statement-label constant. Its attributes agree with those of the fourth parameter. But since it is a constant, a dummy argument is created for it. When SUBR is invoked, the dummy argument is passed.

In SUBR, four parameters are explicitly declared in the PROCEDURE statement. If no further explicit declarations were given for these parameters, arithmetic default attributes would be supplied for each. Therefore, since NAME must represent an entry name, it is explicitly declared with the ENTRY attribute, and since TRANPT must represent a statement label, it is explicitly declared with the LABEL attribute. X and J are arithmetic, so the defaults are allowed to apply.

Note that the appearance of NAME in the CALL statement does not constitute a contextual declaration of NAME as an entry name. Such a contextual declaration can be made only if no explicit declaration applies, and the appearance of NAME in the PROCEDURE statement of SUBR constitutes an explicit declaration of NAME as a parameter. If the attributes of a parameter are not explicitly declared in a complementary DECLARE statement, arithmetic defaults apply. Consequently, NAME must be explicitly declared to have the ENTRY attribute; otherwise, it would be assumed to be a binary fixed-point variable, and its use in the CALL statement would result in an error.

USE OF THE ENTRY ATTRIBUTE

If an ENTRY attribute without parameter attribute lists is specified for an identifier, it indicates only that the identifier is an entry name. In this case, the argument and parameter attributes must agree. A contextual declaration of an identifier as an entry name supplies an ENTRY attribute specification of this type.

If an ENTRY attribute specification with parameter attribute lists is supplied for the invoked entry name, each argument is converted, if necessary, to conform to the attributes specified for its corresponding parameter in the ENTRY attribute specification. String lengths and area sizes are considered to match in two circumstances only: if the length or area size is specified by an asterisk in the ENTRY attribute or if declarations for both the argument and the parameter contain the same decimal integer constant.

Dummy arguments are allocated immediately before invocation of the procedure and freed upon return.

The asterisk notation may be used in the ENTRY attribute to specify that for strings, areas, or arrays, the argument length, size, or bounds is to be assumed for the parameter.

Example:

```
A:   PROCEDURE;
     DECLARE B ENTRY (FIXED,),
        (C,D) FLOAT;
           .
           .
           .
     CALL B(C,D);
           .
           .
           .
     END A;

B:   PROCEDURE (P,Q);
     DECLARE P FIXED, Q FLOAT;
           .
           .
           .
     END B;
```

The specification of the ENTRY attribute in procedure A indicates that B has two parameters, the first with attribute FIXED and the second, indicated by the comma, with attributes that match those of the argument. However, the arguments C and D both have the FLOAT attribute. Since C is to be fixed-point when it is passed to procedure B, a dummy argument is constructed by converting C from floating-point to fixed-point. This dummy argument is then passed to B.

CORRESPONDENCE OF PARAMETERS AND ARGUMENTS

If a parameter of an invoked entry is a scalar, the argument must be a scalar expression. The data attributes of the argument or dummy argument must agree with the corresponding attributes of the parameter. If a constant is used to specify the length of a string parameter or the size of an area parameter in the invoked procedure, the value of the length or size expression of the argument must agree with the constant.

If a parameter of an invoked entry is an array, the argument in general must be an array expression with identical bounds and dimensionality. The argument may be a scalar expression so long as an ENTRY attribute is given for the invoked entry, specifying the dimension attribute and bounds expressions for the relevant parameter. In this case, a dummy array argument will be constructed where the value of each element of the array is the value of the scalar expression. The data attributes of the argument must agree with those of the parameter if a dummy has been created. If constants are used to specify the bounds of the parameter in the invoked procedure, the values of the bounds of the array argument must agree with the values of these constants. ALIGNED and UNALIGNED attributes must agree.

If a parameter is a structure, the argument in general must be a structure expression. When a structure description is given for a parameter in an ENTRY attribute specification, a scalar expression may be specified as the corresponding argument. A dummy structure argument will then be constructed where the value of each element of the structure is the value of the scalar expression. The data attributes of the elements of the structure argument must match those of the associated parameter as specified in the invoked procedure. The relative structuring of the argument and the parameter must be the same, although the level numbers need not be identical. ALIGNED and UNALIGNED attributes must agree. Contained strings and arrays with lengths, areas, and bounds specified by constants must agree as described above.

If a parameter is an area, the corresponding argument must be an area expression. If its size is declared by a constant in the invoked procedure, the corresponding argument must have the same size. This applies to areas in arrays and structures.

If a parameter is a cell, the corresponding argument must be a cell variable whose relative structuring is the same as that of the parameter, although the level numbers need not be identical. This also applies to cells in arrays and structures.

If a parameter is a scalar-label variable, the argument must be a scalar-label expression. If a parameter is an array-label variable, the argument in general must be an array-label variable. If an ENTRY attribute is given for the invoked entry in the invoking procedure, and if the appropriate parameter attribute list specifies that the parameter is a label array, then the argument may also be a scalar-label expression; a dummy label array argument will be suitably constructed. A dummy argument is always constructed when the argument is a label constant.

If the argument is a statement label constant, this statement label constant is qualified by an identification of the current invocation of the block containing the label. Any reference to the parameter is a reference to the statement label in that environment.

If a parameter is an entry parameter, the corresponding argument must be an unparenthesized entry name. If an ENTRY attribute specification is given for the invoked entry in the invoking procedure, and if the appropriate parameter attribute list specifies that the parameter is an entry name and specifies further (nested) parameter lists, then the argument may also be a generic name or the name of a mathematical generic built-in function; the alternative whose parameter attribute list matches the nested parameter list is selected and passed to the parameter.

If a parameter is a pointer-variable, the corresponding argument must be a locator expression. If the argument is an offset variable its value is converted to pointer using the area named in its offset attribute; this offset attribute must specify an area variable, and the parameter must be described as a pointer in the entry attribute. If the argument is an offset function reference, its value is converted to pointer using the area variable named in the offset attribute within the RETURNS attribute in the declaration of the function's name; this offset attribute must specify an area variable, and the parameter must be described as a pointer in the entry attribute.

If a parameter is an offset-variable, the corresponding argument must be a locator expression. If the argument is a pointer expression, an offset-attribute specifying an area variable must be used to describe the parameter in the entry attribute; this area variable is used to convert the pointer expression to area. If the argument is an offset-expression the area variable, if any, associated with the argument and the area variable, if any, in the offset attribute in the entry attribute have no effect on argument passing; if different variables are specified this does not, of itself, cause the creation of a dummy.

If a parameter is a file parameter, the argument must be a file name or parameter. With the exception of FILE, any file attributes declared for the parameter are ignored.

ALLOCATION OF PARAMETERS

A simple parameter, that is, one that is not controlled, may correspond to an argument of any storage class; if more than one generation of the argument exists, however, the parameter is synonymous only with the generation existing at the point of invocation. At least one generation must exist. A controlled parameter, however, always must be presented with a controlled argument; the argument must be an unsubscripted name of controlled data that is not an element of a structure.

When a procedure is invoked without a task option, the parameter is synonymous with the entire allocation stack of the controlled variable. Thus each reference to the parameter is a reference to the current generation of the associated argument. A controlled parameter may be allocated and/or freed in the invoked procedure, thus manipulating the allocation stack of the associated argument.

When a procedure is attached as a task, only the current generation of a controlled argument is available to the new task. The new task can allocate and free subsequent generations, but it cannot free the generation passed to it.

If storage has not been allocated for an argument passed to a controlled parameter declared with the asterisk notation, explicit bounds or length must be declared in an ALLOCATE statement executed before another reference to the parameter in the invoked procedure.

THE SPECIAL PROCEDURE OPTION RECURSIVE

In the PROCEDURE statement for a given procedure, certain special options that characterize the procedure itself may be specified. (For a complete discussion of these options, see "The PROCEDURE Statement.") One of these, which has particular significance, is the option RECURSIVE. When a procedure of a program is reactivated in a task while it is still active in the same task (see "Activation and Termination of Blocks"), the procedure is said to be used recursively. Any procedure used recursively during program execution must be specified with the RECURSIVE option.

CHAPTER 6: DYNAMIC PROGRAM STRUCTURE

PROGRAM CONTROL

Every program, when it is being executed, has a control that determines the order of execution of the statements. For a discussion of their order see "Sequence of Control," in Chapter 8.

Execution of the program is initiated by the operating system invoking the initial procedure at some entry point. Some implementations may require that this entry point be identified by the OPTIONS option in the PROCEDURE statement of the initial procedure. This procedure cannot have CONTROLLED parameters.

PROLOGUES

On entering a block, certain initial actions are performed, e.g., allocation of storage for automatic variables. These initial actions constitute the prologue.

At the beginning of the prologue, the following items are available for computation:

1. The established generation of automatic and defined variables declared outside the block and known within it.

2. Static variables known within the block.

3. Controlled and based variables known within the block, but only those generations that can be accessed by the task executing the block. Note that, for controlled variables, this means only the most recent generation allocated in the task or inherited by the task.

4. Arguments passed to the block.

The prologue makes available for computation all the other variables known within the block as follows:

5. Automatic variables declared in the block.

6. Defined variables declared within the block.

7. Entry and generic names declared within the block.

In making these items available, the prologue may need to evaluate expressions concerned with automatic and defined data. Such expressions may occur specifying lengths, bounds, sizes of areas, and iteration factors, as well as arguments in a CALL option. Expressions of these kinds also occur in RETURNS and ENTRY attribute specifications. These expressions may depend on items of 1, 2, 3 or 4. They may also be dependent on items 5, 6, and 7 under the following circumstances: If an item is referred to in an expression and the allocation or initialization of a second item depends on that expression, then that first item must in no way be dependent on the second item for its own allocation and initialization. Further, the first item must in no way be dependent on any other item that so depends on the second item.

Example:

The following is illegal:

```
DECLARE (A(M) INITIAL (1),
         M INITIAL ((A(I))3)) AUTO;
```

The evaluations must not invoke irreducible functions. The entry invoked with the INITIAL CALL attribute may be irreducible only in that it sets the data being initialized. The sequence in which the evaluations refer to any abnormal data is not defined.

ACTIVATION AND TERMINATION OF BLOCKS

A begin block is said to be activated when control passes through the BEGIN statement for the block. A procedure block is said to be activated when the procedure is invoked at any one of its entry points.

During certain time intervals of the execution of a program, a block may be active. A block is active if it has been activated and is not yet terminated.

There are a number of ways in which a block may be terminated. These are implied by the following rules:

1. A begin block is terminated when control passes through the END statement for the block.

2. A procedure block is terminated on execution of a RETURN statement or an END statement for the block. (The END statement implies a RETURN statement; see Chapter 8.)

3. A block is terminated on execution of a GO TO statement contained in the block which transfers control to a point not contained in the block. Any intervening blocks are also terminated.

4. The execution of a STOP statement causes termination of the major task.

5. The execution of an EXIT statement causes termination of the task containing the statement and all tasks attached by this task. Thus, all blocks corresponding to these tasks are terminated.

6. When a block B is terminated, all of the dynamic descendants of B also are terminated.

7. When a block is terminated, all active subtasks created during the execution of that block are terminated.

DYNAMIC DESCENDANCE

If a block B is activated and control stays at points internal to B until B is terminated, no other blocks can be activated while B is active. (This discussion is not applicable to the multi-task, or asynchronous, mode of operation, which implies more than a single control.)

However, another block, B1, may be activated from a point internal to block B while B still remains active. This is possible only in the following cases:

1. B1 is a procedure block immediately contained in B (the label of B1 is internal to B) and reached through a procedure reference.

2. B1 is a begin block internal to B and reached through normal flow.

3. B1 is a procedure block not contained in B and reached through a procedure reference. (B1, in this case, may be identical to B, i.e., a recursive call. However, it is to be regarded dynamically as a different block.)

4. B1 is a begin block or a statement specified by an ON statement (see "The ON Statement"), and reached through an interrupt. (For present purposes, even if B1 is a statement, it can be regarded as a block, and this case is dynamically similar to case 1 or case 3 above.)

In any of the above cases, while B1 is active, it is said to be an an <u>immediate dynamic descendant</u> of B.

Block B1 may itself have an immediate dynamic descendant B2, etc., so that a chain of blocks (B, B1, B2,...) is created, where, by definition, all of the blocks are active. In this chain, each of the blocks B1, B2, etc., is said to be a <u>dynamic descendant of B</u>.

It is important for the programmer to note that the termination of a given block may automatically imply the termination of other blocks and that these blocks need not necessarily be contained in the given block; storage for all AUTOMATIC variables declared in these blocks will be released at the time of termination (see "Storage Classes").

DYNAMIC ENCOMPASSING

If block B is a dynamic descendant of block A, then block A <u>dynamically encompasses</u> block B, and block B is <u>dynamically encompassed</u> by block A.

THE ENVIRONMENT OF A BLOCK ACTIVATION

A block is said to <u>statically contain</u> those blocks that are nested within it; the scope of declarations within a block, B, includes those blocks statically contained in B. Now, certain attributes are evaluated and certain generations established on entry to a block; the relevant attributes and generations are:

Generations of automatic data

Generations of simple parameters

Bounds, string-lengths, and area sizes of defined data

Bounds, string lengths, and area sizes within simple parameter attribute lists of entry attributes

String lengths and area sizes within RETURNS attribute specifications and in PROCEDURE and ENTRY statements

When several activations of B are in existence, as in recursion, it is essential to know which activation of B holds the storage and evaluated attributes of data declared in B and known to a given descendant activation of a block statically contained in B. If a block, B1, is nested within n statically containing blocks, the particular activation of each of the n blocks that hold the evaluated attributes and generations known to B1 form the environment of the activation of B1.

The immediate environment of an activation of a begin block is provided by the activation of the immediate statically containing block that activates the begin block.

The immediate environment of an activation of a procedure by one of its entry names (i.e., not by an entry parameter) is provided by the activation of the immediate statically containing block that activates the procedure.

When an entry name is passed as an argument, the immediate environment to be used in subsequent invocations by an entry parameter is determined and passed with it. This environment is provided by the activation, in the current environment of the block that passes the entry name, of the block that statically contains the procedure whose entry name is passed.

The immediate environment of an activation of an on-unit is provided by that activation of the block, containing the ON-statement, in which the on-unit is established.

The immediate environment of an activation of some block, BA, is provided by an activation of the block, B1, which statically contains BA. If BA is nested within the n blocks B1, B2 ... Bn, there is a sequence of block activations such that the activation of B_i+1 provides the immediate environment of the activation of B_i. This sequence provides the complete environment of the activation of BA.

THE ENVIRONMENT OF A LABEL CONSTANT

A label constant written as a label prefix designates a point within the text of a block, B. During execution, there may be several activations of B; it is essential to know to which such activation of B a reference to the label refers.

A reference to a label constant, L, made in some activation of a block B1 is to L in that activation of B which forms part of the current environment of the activation of B1. (Of course, if B and B1 are the same block, L refers to the current block.) When a label-constant is assigned to a label variable, this environmental information is assigned as well; subsequent GO TO statements naming the label variable will reestablish the environment assigned to the variable, and hence may cause blocks to be terminated.

GENERATION OF A VARIABLE

A level-one generation, or allocation, of a variable is created whenever storage is allocated for the variable. A level-one generation, or a subgeneration as described below, consists of the storage for the generation and has associated with it a pointer to the generation and the evaluated set of attributes of the generation. The pointer to the generation serves as a unique identification of the generation. The evaluated set of attributes is established when the generation is allocated and enables the contents of the storage to be interpreted.

In the case of static, automatic, and controlled generations, the pointer to the generation can only be obtained by supplying the variable as the argument of the ADDR built-in function. For based variables a locator variable is specified when a based generation is to be created by an ALLOCATE, LOCATE, or READ statement; a value is assigned to the locator variable enabling it to be used to access the generation that is created.

The storage for a generation contains the values of the various fields in the variable. The evaluated set of attributes of a generation comprises the structuring of the variable, its ALIGNED or UNALIGNED attribute, the data types of its components, and the bounds of arrays, lengths of strings and sizes of areas as evaluated at the point of allocation.

A generation of an aggregate or area variable consists of a number of subgenerations. If a generation is an array, each subscripted item in the array is a subgeneration. If a generation is a structure, each item immediately contained within the structure is a subgeneration. An aggregate subgeneration itself contains further subgenerations. An area generation contains a set of subgenerations corresponding to the generations that have been allocated in the area but not freed. If a subgeneration is an area, the attributes of its subgeneration are significant only if one of these subgenerations is being accessed.

Offset variables may be used to identify the position of a generation within an area. The position is not qualified by the area itself, so the offset may be applied to any suitable area. This is achieved by supplying the offset and the area as arguments of the POINTER built-in function; the result is a pointer identifying the generation within the area.

ALLOCATION OF DATA AND STORAGE CLASSES

Because the internal storage of any computer is limited in size, the efficient use of this storage during the execution of a program is frequently a crucial consideration. The simple static process of data allocation used by many compilers -- the assignment of a distinct storage region for each distinct variable used in the source program -- may be wasteful. Multiple use of a storage region for different data during program execution can reduce the total amount of storage required.

Provisions are included in the language to give the programmer virtually any degree of control over the allocation of storage for the data variables in a program if he chooses to do so.

DEFINITIONS AND RULES

Storage is said to be allocated for a variable when storage is associated with the variable. Allocation for a given variable may take place statically, before execution of the program, or dynamically, during execution.

Storage may be allocated dynamically for a variable and subsequently released. Thus, this storage is freed for possible use in later allocations. If storage has been allocated for a variable and not subsequently released, the variable is said to be in an allocated state.

When a variable appears in a statement of a source program, the appearance is called a reference. If a reference corresponds either to the assignment of a value to the variable (e.g., an appearance on the left side of an assignment statement) or to a use of the value of the variable (e.g., appearance in an expression to be evaluated) the variable must be in an allocated state.

STORAGE CLASSES

Every variable in a program must have a storage class, which specifies the manner of storage allocation.

There are four storage classes. The storage class is specified by declaring the variable with one of the four storage class attributes STATIC, AUTOMATIC, CONTROLLED, or BASED. The storage class may be declared explicitly or by default.

The Static Storage Class

Storage for a variable with the attribute STATIC is allocated before execution of the program and is never released during execution.

The scope attribute of a static variable may be INTERNAL or EXTERNAL. An external variable with unspecified storage class has, by default, the STATIC storage class attribute.

The Automatic Storage Class

If a variable has the attribute AUTOMATIC, the activation and termination of the block containing the declaration of this variable determines storage allocation for the variable. Whenever this block is activated during execution of a program, storage will be allocated for the variable, and the variable will remain in an allocated state until termination of this block. At the time of termination, the storage is released. Thus, the time interval during which the variable is in an allocated state will necessarily include the intervals when the variable is known.

Termination of a block by means of a GO TO, STOP, or EXIT statement may imply simultaneous termination of other blocks and, consequently, simultaneous release of storage for all automatic variables declared in these blocks.

If the block is activated recursively (reactivated one or more times before return), the previous generation of an automatic variable or parameter is "pushed down" on each entrance and "popped up" on each return to yield the proper generation of storage for the variable after each return, until the final return out of the procedure.

Note: The terms "pushed down" and "popped up" refer to the notion of a push-down stack. A push-down stack is a logical device S, similar in behavior to a physical stacking process. When an element is placed in S, it is conceptually placed on top of the elements already in S, which are "pushed down." At any time, if S is not empty, the top element -- the element most recently placed in S -- can be removed from S, and the remaining elements are "popped up."

The scope attribute of an automatic variable must be INTERNAL. An internal variable with unspecified storage class has, by default, the AUTOMATIC storage class attribute.

The Controlled Storage Class

The ALLOCATE statement may specify one or more controlled variables, each with certain optional attributes. Execution of the statement causes the allocation of storage for the variables specified.

The FREE statement may specify one or more controlled variables, and execution of the statement causes the storage most recently allocated for the variables to be released.

At some point in a program, it may not be known whether a controlled variable X is in an allocated state. The built-in function ALLOCATION is provided to test this state. The function reference ALLOCATION (X) will return the value '1'B if any generation of X is in an allocated state, and the value '0'B if not.

More than one ALLOCATE statement specifying the same variable, without an intervening FREE statement creates a push-down stack of generations of that variable. A FREE statement always frees the topmost generation.

Generations that are not explicitly freed are freed automatically upon termination of the task in which they are allocated.

The scope attribute of a controlled variable may be INTERNAL or EXTERNAL.

Example:

```
A:  PROCEDURE;
    DECLARE X STATIC;
        .
        .
        .
    B:  PROCEDURE;
        DECLARE Y (100) CONTROLLED,
            Z CHARACTER (1000);
            .
            .
            .
        ALLOCATE Y;
            .
            .
            .
        FREE Y;
            .
            .
            .
        C:  BEGIN;
            DECLARE Z (100);
                .
                .
                .
            END C;
            .
            .
            .
        RETURN;
            .
            .
            .
        END B;
        .
        .
        .
    END A;
```

Assume in the above example that the termination of procedure A occurs on the return implied by END A, the termination of procedure B occurs on the RETURN statement, and the termination of block C occurs at END C. Then in this example:

Storage for the static variable X is allocated before execution and is never released.

The character-string variable Z is AUTOMATIC by default. Storage is allocated for this Z on entry to procedure B and is released on execution of the RETURN statement.

The array-variable Z is AUTOMATIC by default. Storage is allocated for this Z at the beginning of execution of block C and is released at END C.

Storage for the CONTROLLED variable Y is allocated on execution of the ALLOCATE statement and is released on execution of the FREE statement. After execution of the FREE statement, the variable Y presumably is not used, but the character-string variable Z can be used, since storage is not released for this variable until the termination of procedure B.

The Based Storage Class

The BASED Attribute specifies that generations of the declared variable may be allocated under the control of the programmer. A based variable can be allocated by use of the ALLOCATE statement (optionally in a specified area) and freed by use of the FREE statement. A based variable can be allocated in a buffer by use of the LOCATE statement; such a generation is freed when the record is transmitted by a subsequent LOCATE or WRITE statement for the same file, or when the file is closed. A based generation may also be allocated by a READ statement with the SET option. All based generations allocated in a task, with the exception of those allocated in areas, are automatically freed when the task is terminated.

A based reference comprises two parts which together enable a generation to be accessed. Firstly, there is a based variable which provides the attributes of the generation. Secondly, there is a locator qualifier which identifies the generation; this qualifier is obtained from the based attribute of the based variable unless a qualifier is specified in the reference, in which case it overrides any qualifier given in the based attribute.

| When a BASED variable is used to access a generation, the ALIGNED and UNALIGNED attributes of the BASED variable and the accessed generation must agree.

Based variables need not be allocated. Based references may be used to access generations in any storage class. The ADDR built-in is used to obtain a pointer value which will identify a non-based generation. A based reference refers to an allocated generation if its locator qualifier has a defined value.

ASYNCHRONOUS OPERATIONS AND TASKS

PL/I provides facilities for execution of a program as a set of asynchronous tasks. These facilities include provision for:

1. Creating and terminating tasks

2. Synchronizing tasks

3. Testing whether or not a task is complete

4. Changing the priority of a task

5. Testing the status of the termination of a task

SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS

Unless the program specifies the creation of tasks, the execution of the statements of the program will proceed serially in time, according to the sequence designated by the order of the statements and the control statements. Such operation is said to be synchronous.

In addition to full facilities for conventional synchronous processing, means are provided for performing operations asynchronously.

Some reasons for considering the use of asynchronous operations are:

1. The programmer may wish to make use of computer facilities which can operate simultaneously, e.g., input/output channels, multiple central processing units.

2. A program may be written in which input/output units initiate or complete transmission at unpredictable times, e.g., disc operations, terminals.

The following two diagrams distinguish between synchronous and asynchronous operations. The first diagram depicts the serial action of synchronous operations, and the second diagram depicts the parallel action of asynchronous operations. (The circles represent statements.)

```
 ---o--o---o-----------------o---------
time-->
```

```
                    r--o---o-------...
                    |
                    |
            ro------o----o---------...
            |
            |
   o-o-oL--------o-------o-------o-----------
   time-->
```

In asynchronous operation, once a new line has been started, the statements on that line are executed in sequence, but independently of the statements on any other line. Statements on any two lines need not necessarily be executed simultaneously -- whether this occurs depends on the resources and state of the system.

SYNCHRONIZING TWO ASYNCHRONOUS OPERATIONS

In order that the result of an asynchronous operation may be made available to other tasks, a WAIT statement can be used to synchronize two or more asynchronous operations.

The following diagram illustrates this:

```
  A  B  C  D             E          F   G
  o--o--o--o-------------o----------o---o--...
                         |
time-->                  |
                         |
 -------o--o.........----o----o--------o----------
        L  M             N    O        P
           Wait
```

Assume that before statement N can be executed, both L and E must have been executed. M therefore issues a WAIT statement which will suspend operation on that line until E has been completed. After N, the statements O, P,..., are executed synchronously, as are the statements F, G,...,.

TASKS AND EVENTS

In PL/I, asynchronous operations result from the creation, by the programmer, of tasks or from the initiation of DISPLAY statements or record transmission with an event option. The synchronizing of operations is obtained by waiting on events.

A task is an identifiable execution of a set of instructions. A task is dynamic, and only exists during the execution of a program or part of a program.

A task is not a set of instructions, but an execution of a set of instructions. The instructions themselves, as written by the programmer, may in fact be executed several times in different tasks.

It is necessary for at least one task to exist when a PL/I program is executed. Thus when an external procedure is first entered, its execution is part of a task. This particular task is called the major task; it is created by the operating environment and its creation does not necessarily concern the PL/I programmer. If the programmer is concerned with only synchronous operations, then the major task will be the program itself.

In order to initiate asynchronous operations of tasks, the programmer has to create new tasks, as described below. All tasks created by the programmer are called sub-tasks.

With each task, except the major task, it is possible to associate a task variable. The task variable may be used to refer to and set the priority of the task; it cannot be used, however, to test completion of the task.

A task may be suspended by the programmer until some point in the execution of another task has been reached. The specified point is known as an event and the record of its completion is contained in an event variable and accessed by the COMPLETION built-in function. The value '1'B indicates the event is complete; '0'B indicates the event is incomplete. An event variable also has a status value, accessed by the STATUS built-in function, which indicates the manner in which the event has been completed.

An event variable may be associated with the completion of a task. It is necessary to specify such an event variable if the programmer wishes to synchronize a point in one task with the completion of another task, by means of the WAIT statement.

The DISPLAY statement and some RECORD input/output operations can be associated with event variables. These event variables can then be used in WAIT statements to synchronize the task with the completion of the input/output event.

An event variable remains associated with an event until the event has been completed. During this period of association, the event variable is said to be active. Any attempt to associate it with another event or to modify its completion value is an error, and the ERROR condition is raised.

An event variable associated with a task is set complete when the task is terminated. If the task is terminated by a RETURN or END statement, the status value indicates normal termination; otherwise, the status must have a non-zero value.

On execution of an input/output statement with the EVENT option, the event variable is first set active and then incomplete. This is done before any input/output transmission is initiated, but after any action associated with an implicit opening is complete. An input/output event variable is not set complete until a

WAIT statement naming the associated event variable is executed in the task that initiated the event. The WAIT statement delays execution of this task until any transmission associated with the event has been terminated. If no input/output conditions are to be raised for this operation, the event variable is set complete and is no longer active. If any input/output conditions are to be raised, all conditions are raised during the execution of the WAIT statement. On a normal return from the last on-unit entered as a result of these conditions, or on an abnormal return from one of these on-units, the event variable is set complete and is no longer active (see "Multiple Interrupts" in Appendix 3 for more information).

THE CREATION OF TASKS

In PL/I tasks are created by execution of a CALL statement that contains one or more of the following:

A TASK option
An EVENT option
A PRIORITY option

The called procedure will then be executed asynchronously with the calling procedure. The CALL statement itself is not part of the newly-created task. The execution of the calling procedure is known as _the attaching task_. The execution of the called procedure is known as _the attached task_.

The TASK option is given in order to name the task created by the CALL. This is necessary if the programmer wishes to examine or change the priority of the called procedure, since the PRIORITY function and pseudo-variable have a task name as an argument.

The EVENT option is given if the programmer wishes to issue a WAIT statement which will wait on the completion of the task created by the CALL.

On execution of a CALL statement with the EVENT option, the event variable, which must be inactive, is set incomplete. The variable becomes active immediately before it is set incomplete. All this is accomplished before control passes to the named task.

The task created by the CALL statement must be given a priority. This priority may be specified in either of two ways:

1. through the PRIORITY option in the CALL statement, or
2. by assignment to the PRIORITY pseudo-variable for the task name prior to the execution of the CALL statement that creates the task using the same task name.

If a task is attached without a specified priority, the priority of the attached task is the same as the priority of the attaching task.

The term "task option" will be used in all later discussions to denote any one of the three options TASK, EVENT, or PRIORITY, or any part of these options, or all three.

TERMINATION OF TASKS

A task may be terminated in one of the following ways:

1. Control for the task reaches a RETURN or END statement for the procedure invoked with a task option.
2. Control for any task reaches a STOP statement.
3. Control for the task reaches an EXIT statement.
4. A block or task from which this task is a dynamic descendant is terminated.

When a task is terminated the following actions take place:

1. All I/O events, which were initiated in that task and which are not yet complete, are set complete and their status value is set to 1 if it is not already non-zero. Their results are not defined.
2. All files, which were opened during that task and are not yet closed, are closed. During this process all I/O conditions are disabled.
3. All records locked by the task, or any of its subtasks, are unlocked.
4. All CONTROLLED variables allocated during the execution of the task are freed.
5. BASED variables allocated in AREAs are freed when the AREA in which they were allocated is freed. All other BASED allocations are freed when the task in which they were allocated is freed.
6. All active blocks in the task are terminated. This involves the termination of all tasks initiated during

the execution of these blocks and still active.

7. If the task is terminated by any statement other than a RETURN or END statement in this task, the status value of the event variable associated with the task is set to 1 unless it is already non-zero. In all cases the completion value of the event is set to '1'B.

Variables which were being assigned to at the time of task termination, or data sets associated with OUTPUT or UPDATE files which were being created or updated at the time of task termination, may not have defined values after termination. It is the responsibility of the programmer to ensure that assignment to variables or transmission to files is properly completed before the task performing these operations terminates.

DYNAMIC DESCENDANCE OF TASKS

If, within the execution of a task, a block B is activated and control for that task stays at points internal to B until B is terminated, no other blocks can be activated within that task while B is active.

It is possible, however, for control of that task to pass outside B and cause activation of other blocks while B is still active for single tasking applications in any of the ways described under "Dynamic Descendance." It is also possible for a new control of a task to be initiated during the activation of B by a CALL with a TASK, EVENT or PRIORITY option. Just as all additional blocks activated in the original task are dynamic descendants of B, all blocks in the new task are dynamic descendants of B and of the blocks of which B is a descendant. Most of the rules associated with dynamic descendance apply across task boundaries, e.g. ON units established prior to the attaching of a task are inherited by the subtask just as if the initial block of the subtask had been synchronously called.

Sharing of Data between Tasks

The rules of scope for names apply to blocks whether or not the blocks are invoked as, or by, subtasks. The same variables, or generations of these variables, can therefore be referenced by two or more asynchronously executing tasks. This can give rise to unpredictable or undefined results unless special steps are taken in the source program to ensure that more than one reference to the same variable cannot be in effect at one instant (e.g. by forcing temporary synchronisation by use of WAIT), or unless none of the references to a variable that can be in effect at one instant can assign to the variable. Subject to this qualification and the normal scope rules, the following additional rules apply.

1. Any generation of any variable of any storage class can be referenced in any task by means of an appropriate BASED variable reference. It is the user's responsibility to ensure the required variable is in an allocated state at the time of reference. BASED variables allocated in an AREA are freed when the AREA is freed; all other BASED variables are freed when the task, in which they were allocated, is terminated.

2. Static variables may be referenced in any task in which they are known.

3. Automatic variables can be referred to by any block dynamically descendent from the block which allocates them, regardless of task boundaries.

4. Controlled variables can be referenced in any task in which they are known; however, not all allocations are known in each task. When a task is initiated, only the allocation of each controlled variable currently known by the attaching task is passed to the attached task. Both tasks may reference this allocation. Subsequent allocations in the attached task are known only within the attached task; subsequent allocations in the attaching task are known only within the attaching task. A task may only free allocations it has allocated. It is permissible for no allocations of the controlled variable to exist at the time of attaching. It is not permissible for a task to free a controlled allocation shared with a subtask if the subtask subsequently attempts to reference the generation. When a task is terminated all allocations of controlled storage made within that task are freed.

Sharing Files between Tasks

A file is shared between a task and its subtask if the file is open at the time the subtask is attached. The rules concerning such shared files are as follows.

1. If a subtask shares a file with its attaching task, the subtask must not close the file. A subtask must not access a shared file after its attaching task has closed the file, even if the attaching task reopens the file before the subtask accesses it.

2. If a task shares a file with one of its subtasks it may close the shared file, provided the subtask makes no subsequent attempt to access the file.

3. If a file name is known to a task and its subtask, and its associated file was not open when the subtask was attached, then both the task and its subtask may each separately open, access and close the file.

INTERRUPT OPERATIONS

During the course of program execution any one of a certain set of conditions may occur that can result in an <u>interrupt</u>. An interrupt operation causes the suspension of normal program activities, in order to perform a special action; after the special action, program activities may or may not resume at the point where they were suspended.

For conditions recognized by PL/I, the special action to be taken when an interrupt occurs may be specified by the programmer. To do this, he may specify the condition in an ON statement; therefore these conditions are known as the <u>ON-conditions</u>. A complete list and description of the ON-conditions can be found in Appendix 3. With one exception (see "Programmer Defined ON-Conditions," in this chapter), each ON-condition is named with a unique identifier suggestive of the condition (e.g., ZERODIVIDE specifies the condition obtaining whenever an attempt is made to divide by zero). This collection of names is an intrinsic part of the language, but the names are not reserved; the programmer may use them for other purposes, so long as no ambiguity exists.

PURPOSE OF THE CONDITION PREFIX

In general, during the execution of a statement, an ON condition may be in either an enabled or disabled state.

If a particular condition is enabled and an interrupt occurs during execution of the statement, the action specification for the condition is executed. This action specification may either be standard system action or it may have been specified by the programmer through the use of an ON statement.

If a particular condition is disabled during execution of a statement, it is assumed that the condition will not occur. The result is unpredictable for a statement in which a disabled condition occurs. However for the CHECK condition, results are defined.

By means of condition prefixes, the programmer can control the enabled/disabled status of the following ON conditions:

CHECK	SIZE
CONVERSION	SUBSCRIPTRANGE
FIXEDOVERFLOW	UNDERFLOW
OVERFLOW	ZERODIVIDE
STRINGRANGE	

The appearance of any of the above keywords in a prefix list causes the associated condition to be enabled for the scope of the prefix. The appearance of any of the above preceded by a NO (with no separating blank) causes the associated condition to be disabled for the scope of the prefix.

SCOPE OF THE CONDITION PREFIX

The <u>scope</u> of the prefix depends upon the statement to which it is attached.

If the statement is a PROCEDURE or BEGIN statement, the scope of the prefix is the block defined by this statement, including all nested blocks, except those blocks and statements for which the condition is respecified. The scope does <u>not</u> include procedures that lie outside the scope as defined above but which may be invoked by the execution of statements in this scope. The identifier list of a CHECK prefix to a PROCEDURE or BEGIN statement belongs to the scope of the corresponding procedure or begin block. If a variable in this list is redeclared in a nested block, it is no longer in the "checked" state, unless, of course, it appears in a CHECK prefix for that nested block. This does not apply however, if both declarations refer to the same external name.

If the statement is an IF statement or an ON statement, the scope of the prefix does not include the blocks or groups that are part of the statement. Any such block may also have an attached prefix, whose scope rules are implied by the other rules given here.

For any other statement, the scope of the prefix is that of the statement itself, including any expressions evaluated during the execution of the statement but not any procedure explicitly called by the statement.

The CHECK Condition

The CHECK condition is provided for program testing. The keyword CHECK in a prefix list is followed by a parenthesized name list. The names in the list may be statement label constants, entry names, and variables, including array and structure variables and label variables. Subscripted names are not allowed, but qualified names can be used.

The CHECK prefix may be attached only to PROCEDURE or BEGIN statements, and therefore, it always applies to an entire block.

An interrupt will generally occur immediately after the execution of a statement in which the value of a variable in a check list may have been altered. With statement labels and entry names, however, the interrupt occurs immediately before the execution of the statement or the invocation of the entry name.

The system action for the CHECK condition is to print the identifier causing the interrupt and, if it is a variable (other than program control data), to print its new value in the form of data-directed output on a debugging file. For program control data, only the variable is printed; no value is included.

USE OF THE ON STATEMENT

In order to define the action to be taken when an interrupt occurs, the programmer may write an ON statement. See "The ON Statement," Chapter 8, for the general form of the statement, the syntax and other details.

When an ON statement that is internal to a given block (for example, a block B) is executed, it causes a preparatory action with the following effect:

> If, during the execution of any statement after the execution of the ON statement and before the termination of block B (including the execution of statements in all dynamic descendants of block B), the condition specified in the ON statement ever occurs and an interrupt results, the statement or begin block specified in the ON statement will be executed as though it were invoked as a procedure block. (If SNAP also has been specified, a standard action providing program checkout information will precede this pseudo-invocation.) Control normally will be returned to the point of interrupt or to the statement following the one that was interrupted.

When an ON statement specifying a given condition is executed, the action to be taken is established by the execution. The time interval during which this on-unit is effective is defined above in the description of the effect of an ON statement. There are two qualifications to this description:

1. If, after a given action is established by execution of an ON statement, and while this on-unit is still effective, another ON statement specifying the same condition is executed, then this latter ON statement will take effect as described above, so that its specified action will determine the interrupt action for the given condition. (The effect of the old ON statement is either temporarily suspended or completely nullified, depending upon whether or not the new ON statement is in a block dynamically descendant from the block to which the old ON statement is internal; see "The ON Statement" and "The REVERT Statement" for more details.)

2. There are nine ON-conditions whose names (possibly preceded by the word "NO" without intervening blanks) may appear in a condition prefix. Even when one of these conditions appears in an ON statement, occurrence of the condition will not necessarily result in an interrupt. For an interrupt to occur, there are certain additional requirements, which are described in the following paragraph.

   There are four of these nine ON-conditions, SIZE, SUBSCRIPTRANGE, STRINGRANGE, and CHECK (identifier list), for which an interrupt will not take place when the condition occurs unless the programmer specifically designates that the interrupt _is_ to take place. He may enable this condition by explicitly specifying the condition in a prefix whose scope will cover the calculation where the condition may occur. If a calculation resulting in the occurrence of either of these conditions does not lie within the scope of such a prefix, no interrupts will occur. The other five of these nine ON-conditions, namely OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and FIXEDOVERFLOW, are always enabled, but the programmer may specifically designate that an interrupt is not to take place. An interrupt for any one of these conditions will always take place when the condition occurs unless the occurrence is in a calculation lying within the scope of a prefix specifying NOOVERFLOW, NOUNDERFLOW, NOZERODIVIDE, NOCONVERSION, or NOFIXEDOVERFLOW, respectively.

The other conditions cannot be named in prefixes, but they are always enabled and cannot be disabled.

SYSTEM INTERRUPT ACTION

Each of the ON-conditions has a standard action defined for it if an interrupt should occur. If no established on-unit is in force for a given condition at the time that condition is raised and causes an interrupt, then standard system action will be taken. Standard system action is dependent upon the nature of the condition. If the programmer does not want the system action in the case where one of these conditions may occur and cause an interrupt, he must specify an alternative action for the condition through use of the ON statement.

In some situations, the programmer may want to specify his own action for a given condition, to have it hold for part of the execution of the program, and then to have this specification nullified and allow the standard system action. In this case, he may use the keyword SYSTEM, as follows:

```
    ON condition-name SYSTEM;
```

Example 1:

```
A:  PROCEDURE;
      .
      .
      .
    ON OVERFLOW
            BEGIN;
            DECLARE NUMBOV STATIC
              INITIAL (0);
            NUMBOV=NUMBOV + 1;
            IF NUMBOV = 100 THEN GO
              TO OVERR;
            END;
      .
      .
      .
    ON OVERFLOW;
      .
      .
      .
    ON OVERFLOW SYSTEM;
      .
      .
      .
    END A;
```

In the above example, assume that the program consists only of procedure A, that the three ON statements are the only ON statements involving the OVERFLOW condition, that they are internal to procedure A, and that they are executed in their physical order.

When program execution begins, the OVERFLOW condition is enabled by the system; any floating-point overflow condition that occurs before the first ON OVERFLOW statement is executed will result in an interrupt, with standard system action. However, the execution of the first ON OVERFLOW statement establishes the action specified in the BEGIN block. (The number of overflows is counted and if this number has not reached 100, the action is finished.) Any OVERFLOW interrupts will receive this action until the second ON OVERFLOW statement is executed. The action specified here is a null statement; any subsequent OVERFLOW interrupts will effectively be ignored until control reaches the third ON OVERFLOW statement, which reestablishes the standard system action.

Example 2:

```
(SIZE): A:  PROCEDURE;
              .
              .
              .
            ON SIZE GO TO AERR;
              .
              .
              .
            CALL B;
              .
              .
              .
            END A;
(SIZE, NOOVERFLOW):  B:  PROCEDURE;
              .
              .
              .
            ON SIZE GO TO BERR;
              .
              .
              .
            RETURN;
            END B;
```

In the above example, the prefix (SIZE) enables that condition for procedure A and specifies that if a SIZE condition occurs during any calculation in procedure A, an interrupt is to take place. The prefix (SIZE, NOOVERFLOW) for procedure B specifies the same requirement with respect to a SIZE error for procedure B; in addition, it specifies for procedure B that any interrupt that might be caused by an OVERFLOW condition is to be suppressed.

After the beginning of execution of procedure A, and before the execution of the first ON statement, any SIZE condition will result in an interrupt with standard system action. After execution of this ON statement, and before execution of the ON statement in the invoked procedure B, any SIZE condition will result in an interrupt with the action GO TO AERR. After execution of the ON statement in procedure B, the action GO TO BERR becomes established for the SIZE condition, but the effect of the previous ON statement is suspended only temporarily. After the RETURN statement in procedure B is executed, the effect of this previous ON statement is reinstated, so that SIZE conditions occurring after this point again result in the action GO TO AERR.

If any floating-point overflow condition occurs during the execution of procedure A, an interrupt will result with the standard system action for the OVERFLOW condition. However, for any occurrence of an OVERFLOW condition during the execution of procedure B, the interrupt will be suppressed.

Example 3:

```
X: PROCEDURE;
   DECLARE A,B;
   ON OVERFLOW BEGIN;
                PUT DATA (A,B);
                END;
   .
   .
   .
Y: BEGIN;
   DECLARE A,B;
   .
   .
   END Y;
   .
   .
   .
   END X;
```

This example illustrates the effect of establishment of the generation of variables at the time an ON condition is executed. If the OVERFLOW condition should arise, the values transmitted by the PUT statement in the on-unit will be the values of the variables A and B that are declared in the outer block. This is true, even if the OVERFLOW condition should arise during execution of the begin block Y, where A and B have been redeclared.

USE OF THE REVERT STATEMENT

The REVERT statement may be used, following an ON statement, to reinstate an action specification that existed in the immediate, dynamically encompassing block at the time the descendant block was invoked.

Example:

```
(SIZE): A: PROCEDURE;
           ON SIZE GO TO AERR;
            .
            .
            .
           CALL B;
            .
            .
            .
           END A;
(SIZE): B: PROCEDURE;
           ON SIZE GO TO BERR;
            .
            .
            .
           REVERT SIZE;
            .
            .
            .
           END B;
```

In the above example, if a SIZE condition occurs in procedure B after execution of the ON statement, an interrupt will take place with the resulting action GO TO BERR. After execution of the REVERT statement, the condition as specified by the ON statement in procedure A is reinstated. Program control remains in procedure B, but any subsequent SIZE condition that occurs in procedure B will cause an interrupt with the action GO TO AERR.

PROGRAMMER-DEFINED ON-CONDITIONS

An identifier can be used to create a condition name by means of the keyword CONDITION used in the ON statement, as follows:

```
ON CONDITION(identifier) on-unit
```

Such a statement contextually declares the identifier to be a condition-name and the execution of the statement provides an on-unit. The condition can be caused to occur only by the execution of a SIGNAL statement (see "The SIGNAL Statement").

For example, if the following statement is executed:

```
ON CONDITION(ABC) block
```

and later the following statement is executed:

```
SIGNAL CONDITION(ABC);
```

then the latter execution will (by definition of the SIGNAL statement) cause an interrupt, with the action defined by the block in the ON statement.

CONDITION BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES

The condition built-in functions are provided for the investigation of interrupts. Each such function is associated with certain conditions:

Built-in Function	Associated Conditions
ONFILE	I/O conditions, and CONVERSION raised during an I/O operation
ONLOC	All conditions
ONSOURCE	CONVERSION condition
ONCHAR	CONVERSION condition
ONKEY	I/O condition or CONVERSION condition raised by an operation on a KEYED file
ONCODE	All conditions
DATAFIELD	NAME condition
ONCOUNT	I/O conditions for an I/O operation with the event option

Appendix 1 gives the value returned by these functions when they are used in the following contexts:

1. An on-unit for one of the associated condition for the function, or an ERROR on-unit entered as standard system action for one of the associated conditions. However, if the condition is CONVERSION and this on-unit is entered via a SIGNAL statement, ONCHAR is the character blank and ONSOURCE is the null string.

2. A block, B, which is a dynamic descendant of such an on-unit, provided that no intervening block is an on-unit for one of the conditions associated with the function nor is any such block an ERROR on-unit entered as standard system action for, or normal return from, any of the conditions associated with the function.

In all other contexts the value of a condition built-in function is the null character string, except for ONCHAR which is the single character blank and ONCODE and ONCOUNT which have the value zero.

The condition built-in functions are inherited by blocks in a manner analagous to the inheritance of on-units. Thus if, for example, the CONVERSION condition occurs in a CONVERSION on-unit, the values of ONSOURCE and ONCHAR are stacked before the new CONVERSION on-unit is entered. Within the new CONVERSION on-unit ONCHAR and ONSOURCE have values determined by this second conversion interrupt; the values pertaining to the first interrupt are re-established when control returns from the second on-unit.

The condition pseudo-variables, i.e., ONCHAR and ONSOURCE, may be used within blocks defined in 1 and 2 above to alter a character string value whose conversion has raised a CONVERSION interrupt. If the source for the conversion is a variable or, pseudo-variable or if it is specified by the SUBSTR built-in function with a variable as its first argument, then within these blocks an assignment to the condition pseudo-variables is an assignment to the variable concerned. It is an error to assign to the condition pseudo-variables in any other blocks.

CHAPTER 7. INPUT/OUTPUT

A collection of data external to the program constitutes a data set. Input activity transmits data from a data set to a program. Output activity transmits data from a program to a data set. Input/output statements refer to a filename declared in the program.

In STREAM input/output, the data set can be considered to be a continuous stream of characters. The GET and PUT statements are used to transmit data values from and to the data set. Conversions may occur during transmission (see "Data Stream Transmission," below).

In RECORD input/output, the data set consists of discrete records. The READ and WRITE statements cause a single record to be transmitted from or to the data set. Transmission is direct, without any conversion, either directly to data variables or to an intermediate buffer that may be addressable. When transmission is to or from data variables, the attributes of the variables should accurately describe the composition of the record.

FILE OPENING AND FILE ATTRIBUTES

The file attributes are discussed in Chapter 4. This section describes how attributes are collected and become associated with a file, as well as describing how a file is opened.

The file attributes can be divided into two categories, alternative attributes and additive attributes. Alternative attributes are those in which one of a group may be selected. If there is no explicit or implied declaration for one of the alternatives, and if one of those alternatives is required, a default attribute is selected. Additive attributes are those that are never applied by default and must always be stated explicitly (except KEYED which is implied by DIRECT), either in a file declaration or in the OPEN statement (the one exception is that PRINT may be applied by default for the SYSPRINT file, see "Standard Files").

Following is a summary of the alternative attributes and their defaults:

Attributes	Default
STREAM|RECORD	STREAM
INPUT|OUTPUT|UPDATE	INPUT
SEQUENTIAL|DIRECT	SEQUENTIAL
BUFFERED|UNBUFFERED	BUFFERED
INTERNAL|EXTERNAL	EXTERNAL

Following is a list of the additive attributes:

PRINT

BACKWARDS

EXCLUSIVE

KEYED

ENVIRONMENT (option-list)

OPENING A FILE

The opening of a file is the means by which a filename is associated with a particular data set. The identity of the data set can be indicated through the TITLE option of the OPEN statement; otherwise, the filename will indicate the identity of the data set. A part of the opening process is the completion of the set of attributes that describe the composition of the data set and the method in which the individual records of the data set will be accessed. A file can be opened either explicitly or implicitly.

Opening a file for stream input, for SEQUENTIAL INPUT forwards, or for SEQUENTIAL UPDATE causes the data set to be positioned to the first record of the data set. Opening for backwards causes the data set to be positioned to the last record.

Explicit Opening

A file is opened explicitly through execution of an OPEN statement that specifies the filename. The OPEN statement may list any of the attributes given above except the ENVIRONMENT, INTERNAL, or EXTERNAL attributes. Attributes listed in an OPEN statement are merged with any attributes listed in a file declaration for that filename. In an explicit opening, the OPEN statement must be executed prior to the execution of any of the statements listed below under "Implicit Opening" that refer to that filename.

Implicit Opening

An implicit opening of a file may occur if one of the statements listed below is executed prior to the execution of an OPEN statement specifying the same filename. The statement type is used to determine the usage and function attributes of the file if they have not been explicitly stated in a DECLARE statement. The effect of an implicit opening, caused by one of these statements, is as if the statement were preceded by an OPEN statement specifying the attributes deduced from the statement type.

Following is a list of the statement identifiers and the attributes that will be deduced from each and that will be applied in the absence of an explicit declaration to the contrary:

Statement Identifier	Attributes Deduced
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT
WRITE	RECORD, OUTPUT
REWRITE	RECORD, UPDATE
LOCATE	RECORD, OUTPUT, SEQUENTIAL, BUFFERED
DELETE	RECORD, DIRECT, UPDATE
UNLOCK	RECORD, DIRECT, UPDATE, EXCLUSIVE

Merging of Attributes

There must be no conflict between the attributes specified in a file declaration and the attributes merged as the result of the file opening, either explicit or implicit. For example, the attributes INPUT and UPDATE are in conflict, as are the attributes UPDATE and STREAM.

After the attributes are merged, the attribute implications, listed below, are applied prior to the application of default attributes discussed earlier in this section. Implied attributes can also cause a conflict. If a conflict in attributes exists after the application of default attributes, the UNDEFINEDFILE condition is raised.

Following is a list of attributes and the other attributes that each implies after merging:

Merged Attribute	Implied Attribute(s)
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD,KEYED
BUFFERED	RECORD, SEQUENTIAL
UNBUFFERED	RECORD, SEQUENTIAL
PRINT	OUTPUT, STREAM
BACKWARDS	RECORD, SEQUENTIAL, INPUT
EXCLUSIVE	RECORD,KEYED, DIRECT, UPDATE
KEYED	RECORD

The following two examples illustrate attribute merging for an explicit opening and for an implicit opening:

Explicit opening example

```
DECLARE LISTING FILE STREAM;
    .
    .
    .
OPEN FILE (LISTING) PRINT;
```

The filename LISTING has the EXTERNAL attribute by default.

Attributes after merge, due to execution of the OPEN statement, are EXTERNAL, STREAM and PRINT.

Attributes after implication are EXTERNAL, STREAM, PRINT, and OUTPUT.

Since this is a complete set of file attributes, no file attribute defaults are applied. The default attribute BUFFERED does not apply as this attribute can be specified only for SEQUENTIAL RECORD files.

Implicit opening example

```
DECLARE MASTER FILE KEYED INTERNAL;
    .
    .
    .
READ FILE (MASTER) INTO
               (MASTER_RECORD)
               KEYTO (MASTER_KEY);
```

Attributes after merge due to the opening caused by execution of the READ statement are KEYED, INTERNAL, RECORD, and INPUT.

Attributes after implication are KEYED, INTERNAL, RECORD and INPUT. There are no additional attributes implied.

Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, SEQUENTIAL, and BUFFERED.

In addition, ENVIRONMENT may be specified with any valid combination, and each filename is external or internal.

Valid Combinations of File Attributes

Valid complete combinations of file attributes are as follows:

```
FILE STREAM INPUT

FILE STREAM OUTPUT

FILE STREAM OUTPUT PRINT

FILE RECORD INPUT SEQUENTIAL BUFFERED

FILE RECORD INPUT SEQUENTIAL BUFFERED
     BACKWARDS

FILE RECORD INPUT SEQUENTIAL BUFFERED
     KEYED

FILE RECORD INPUT SEQUENTIAL BUFFERED
     KEYED BACKWARDS

FILE RECORD OUTPUT SEQUENTIAL BUFFERED

FILE RECORD OUTPUT SEQUENTIAL BUFFERED
     KEYED

FILE RECORD UPDATE SEQUENTIAL BUFFERED

FILE RECORD UPDATE SEQUENTIAL BUFFERED
     KEYED

FILE RECORD INPUT SEQUENTIAL UNBUFFERED

FILE RECORD INPUT SEQUENTIAL UNBUFFERED
     BACKWARDS

FILE RECORD INPUT SEQUENTIAL UNBUFFERED
     KEYED

FILE RECORD INPUT SEQUENTIAL UNBUFFERED
     KEYED BACKWARDS

FILE RECORD OUTPUT SEQUENTIAL UNBUFFERED

FILE RECORD OUTPUT SEQUENTIAL UNBUFFERED
     KEYED

FILE RECORD UPDATE SEQUENTIAL UNBUFFERED

FILE RECORD UPDATE SEQUENTIAL UNBUFFERED
     KEYED

FILE RECORD INPUT DIRECT KEYED

FILE RECORD OUTPUT DIRECT KEYED

FILE RECORD UPDATE DIRECT KEYED

FILE RECORD UPDATE DIRECT KEYED EXCLUSIVE
```

DATA STREAM TRANSMISSION

There are three modes of STREAM transmission: list-directed, data-directed, and edit-directed. All of these modes of transmission utilize data specifications as described in the next section. This section discusses the general characteristics of the transmission modes. The details of these transmission modes are discussed later in the chapter.

LIST-DIRECTED TRANSMISSION

List-directed transmission permits the user to specify the storage area to which data is assigned or from which data is transmitted without specifying the format.

Input: The data in the stream is in the form of optionally signed valid constants or of expressions to represent complex constants. The program storage areas to which the data is to be assigned is specified by a data list.

Output: The data values to be transmitted are specified by a data list. The form of the data placed in the stream is a function of the data value and precision.

DATA-DIRECTED TRANSMISSION

Data-directed transmission permits the user to read or write self-identifying data.

Input: The data in the stream is in the form of optionally signed valid constants and includes information identifying the program storage areas to which the data is to be assigned.

Output: The data values to be transmitted are specified by a data list. The data placed in the stream has the form of constants and includes the name of the data being transmitted.

EDIT-DIRECTED TRANSMISSION

Edit-directed transmission permits the user to specify the storage area to which data is to be assigned or from which data is to be transmitted and the form of data fields in the stream.

Input: The form of the data in the stream is defined by a format list. The program storage areas to which the data is to be assigned is specified by a data list.

Output: The data values to be transmitted are defined by a data list. The form that the data is to have in the stream is defined by a format list.

DATA STREAM DATA SPECIFICATIONS

Data specifications are given in GET and PUT statements to identify the data to be transmitted. The form of the data specifications correspond to the modes of transmission.

DATA LISTS

List-directed and edit-directed data specifications require a data list to specify the data items to be transmitted. A data-directed data specification may or may not include a data list.

General format:

```
    (data-list)
where "data list" is defined as:
    element [, element] ...
```

Syntax rules:

The nature of the elements depends upon whether the data list is used for input or for output. The rules for each are as follows:

1. On input, each data-list element for edit-directed and list-directed data may be one of the following: a scalar name, an array name, a structure name, a pseudo-variable, or a repetitive specification involving any of these elements. For a data-directed data specification, each data-list element may be an unsubscripted scalar, array or structure name.

2. On output, each data-list element for edit-directed and list-directed data specifications may be one of the following: a scalar expression, an array expression, a structure expression, or a repetitive specification involving any of these elements. For a data-directed data specification, each data-list element may be a scalar, array, or structure name, or a repetitive specification involving any of these elements.

3. The elements of a data list must be of arithmetic or string data type.

4. A data list must be enclosed in its own set of delimiting parentheses.

Repetitive Specification

A repetitive specification appears in a data list as follows:

```
(repetitive-specification)
```

General format is shown in Figure 1.

Syntax rules:

1. Each repetitive specification must have its own set of delimiting parentheses, the first preceding the first applicable element, and the second following the applicable DO specification.

```
element [,element]...DO {scalar-variable        } = specification [,specification]...
                        {scalar-pseudo-variable }

A specification has the following format:

expression-1  [TO expression-2   [BY expression-3]]   [WHILE (expression-4)]
              [BY expression-3   [TO expression-2]]
```

Figure 1. General Format for Repetitive Specification

2. Each element in the element list of the repetitive specification is the same as those described for data-list elements above.

3. The expressions in the specification are described as follows:

   a. Each expression in the specification is a scalar expression.

   b. In the specification, expression 1 represents the starting value of the control variable or pseudo-variable. Expression 3 represents the increment to be added to the control variable after each repetition of data-list elements in the repetitive specification. Expression 2 represents the terminating value of the control variable. The exact meaning of the specification is identical to that of a DO statement with the same specification. When the last specification is completed, control passes to the next element in the data list.

4. Repetitive specification may be nested to any depth. That is, each element in the element list may be a repetitive specification. A repetitive specification involving m elements repeated n times is equivalent to m*n elements. For example, consider the following statement:

```
GET LIST (((A(I,J) DO I = 1 TO 2)
     DO J = 3 TO 4));
```

This is equivalent to:

```
DO J = 3 TO 4;
   DO I = 1 TO 2;
   GET LIST (A(I,J));
   END;
END;
```

It gives values to the elements of the array A in the following order:

A(1,3), A(2,3), A(1,4), A(2,4)

Consider the following example:

```
PUT  LIST ((A(J),(B(I,J) DO I=1 TO 10)
   DO J=5 TO 10));
```

This is equivalent to:

```
DO J=5 TO 10;
PUT LIST (A(J));
  DO I=1 TO 10;
  PUT LIST (B(I,J));
  END;
END;
```

Transmission of Data-List Elements

If a data-list element is of complex mode, the real part is transmitted before the imaginary part.

If a data-list element is an array name, the elements of the array are transmitted in row-major order, that is, with the rightmost subscript of the array varying most frequently.

If a data-list element is a structure name, the elements of the structure are transmitted in the order specified in the structure declaration. For example, if the structure declaration was:

```
DECLARE 1 A(10), 2 B, 2 C;
```

then the statement

```
PUT FILE (X) LIST (A);
```

would result in the output being ordered as follows:

```
     A.B(1)  A.C(1)  A.B(2)  A.C(2)  A.B(3)
A.C(3) ....  etc.
```

If, however, the declaration had been:

```
DECLARE 1 A, 2 B(10), 2 C(10);
```

then the same PUT statement would produce:

```
     A.B(1)  A.B(2)  A.B(3)  ....   A.B(10)
A.C(1) A.C(2) A.C(3) ....  A.C(10).
```

If, within a data list used in an input statement, a variable is assigned a value, this new value is used in all later references in the data list, and the format list, if present.

Example:

In the following statement, B is a structure, XSTRING is a character string, and C is an array:

```
DECLARE A FLOAT, 1 B, 2 P, 2 E, 3 F,
  XSTRING CHARACTER (6), C(10) FIXED;
```

The following data list, involving these data items, and the scalar variable A, may be used for input or output:

```
(A,B, SUBSTR (XSTRING, 2),
     (C(I) DO I = 2 TO 7))
```

The data-list elements are transmitted in the following order:

A - the scalar variable is transmitted

P,F - the elements of the structure B are transmitted

SUBSTR (XSTRING, 2) - the second through sixth characters of the string XSTRING are transmitted

C(2), C(3),..., C(7) - the six specified elements of the array are transmitted

LIST-DIRECTED DATA SPECIFICATION

General format:

LIST (data-list)

Syntax rules:

The "data list" is described in the preceding discussion.

List-Directed Input Format

When the data item is an array name and the data consists of constants, the first constant is assigned to the first element of the array, the following constant to the second element, etc., in row-major order.

A structure name in the data list represents a list of the contained scalar variables and arrays in the order specified in the structure description.

Data in the stream has one of the following general forms:

```
⎧[+|-]arithmetic-constant                        ⎫
⎨character-string-constant                       ⎬
⎨bit-string-constant                             ⎬
⎩[+|-]real-constant{+|-}imaginary-constant       ⎭
```

Sterling constants cannot be used. A string constant must be one of the two permitted forms listed above. Iteration and string repetition factors are not allowed.

Constants and complex expressions may be surrounded by blanks, which are not treated as part of the data. However, blanks cannot appear between the optional sign and the constant, nor can they precede the central sign in a complex expression.

Data items in the stream must be separated either by a blank or by a comma. This separator may be preceded and/or followed by an arbitrary number of blanks. A null field in the stream is indicated either by the very first non-blank character in the stream being a comma, or by two adjacent commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated item in the data list specification is to remain unchanged.

The transmission of the list of constants on input is terminated by expiration of the data list or by the end-of-file condition. In the former case, positioning is always at the character following the first blank or comma following the last data item. More than one blank can separate two data items, and a comma separator may be preceded or followed by one or more blanks. In such cases, a subsequent list- or data-directed GET will ignore intervening blanks and the comma (if present), and will access the next data item. However, if an edit-directed GET should follow, the first character accessed will be the character to which the file has been positioned (in other words, the next data item will begin with the first character following the blank or comma that separated it from the previous data item).

If the data is a character-string constant, the surrounding quotation marks are deleted and the enclosed characters interpreted as a character string.

If the data is a bit-string constant, it is interpreted as a bit string.

If the data is an arithmetic constant or complex expression, it is converted to coded arithmetic with the base, scale, mode, and precision implied by the constant.

The list item is then examined and the interpreted string value is assigned to it as shown in Figure 2.

The type conversions are described in Chapter 3, except arithmetic to character conversion which is described below under "List-Directed Output Format."

List-Directed Output Format

The values of the scalar variables in the data list are converted to a character representation of the data value, as described below, and transmitted to the data stream.

In general, a blank is used to separate data items transmitted. However, for PRINT files, implementation-defined tabs are provided such that the printing of a data item

Stream Item	Data List	Conversion
Character string	Arithmetic	Character to Arithmetic
	Character String	Character string assignment
	Bit String	Character to bit string
Bit string	Arithmetic	Bit string to Arithmetic
	Character String	Bit string to Character string
	Bit String	Bit string assignment
Arithmetic	Arithmetic	Arithmetic type conversion
	Character String	Arithmetic to Character string
	Bit string	Arithmetic to Bit string

Figure 2. List-Directed Input Conversion

is always followed by a positioning to the next available tab position. If a numeric data item is longer than the number of characters remaining on the current line, the entire item will be printed starting at the beginning of the next line. (Of course, if the length of the item is greater than the size of the line, splitting must occur.)

The length of the data field placed in the data set is a function of the internal precision and value of the data item.

CODED ARITHMETIC DATA: The external form of coded arithmetic data is a possibly signed valid decimal constant whose field width, *w*, is a function of the internal precision declared for the data item and the value of the data item. In the discussion below, the following symbols are used:

1. The letter *w* represents the field width, which is defined as the length of the data field.

2. The letter *d* represents the number of positions in the external data field to the right of the decimal point.

3. The letter *p* represents the total number of digits in the data item after any necessary conversion to decimal.

4. The letter *q* represents the scale factor of the data item after any necessary conversion to decimal.

5. The letter *s* represents a scaling factor as described for floating-point data.

6. The letters *yyy* represent a scaling factor for fixed-point data. The letter *F* actually appears in the output stream to indicate the presence of a scaling factor. Its value is similar to the value of E in a floating-point number.

7. The letter *x* represents any decimal digit.

8. The letter *b* represents a blank position in the output.

9. The letter *n* represents the number of decimal digits in the exponent, which is defined separately for each implementation.

There are five kinds of coded arithmetic data to consider: coded real fixed-point decimal data, coded real fixed-point binary data, coded real floating-point decimal data, coded real floating-point binary data, and coded complex data.

Note: The discussions below apply to coded arithmetic data only when the value of the data item to be transmitted is greater than or less than zero. If the converted decimal value of a fixed-point item is equal to zero, the following rules apply:

1. If q = 0, the representation transmitted is a single zero preceded by p+2 blanks.

2. If p>=q>0, the representation transmitted is a single zero preceded by p-q+1 blanks, and followed by a decimal point and q zeros.

3. If p<q or q<0, the representation transmitted is a single zero preceded by p blanks and followed by F{+|-} *n* digits.

If the converted decimal value of a floating-point item is equal to zero, the representation transmitted is a single zero, followed by a decimal point, p-1 zeros, and E+*n* zeros.

<u>Coded Real Fixed-Point Decimal Data:</u> A decimal fixed-point source with precision (p,q) is converted to character-string representation as follows:

1. If p>=q>=0 (that is, if the assumed decimal point lies within the field of the internal representation) then:

   a. The constant is right adjusted in a field of width p+3.

   b. Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number.

   c. If the value is negative, a minus sign precedes the first significant digit (or the zero before the point of a fractional number). Positive values are <u>unsigned</u>.

   d. Unless the source is an integer, the constant has q fractional digits. If the source is an integer, there is no decimal point.

2. If q is negative or greater than p, a scaling factor is appended to the right of the constant. The constant itself is of the same form as an integer. The scaling factor has the form:

   ```
   F{+|-}nnn
   ```

   where {+|-}nnn has the value -q.

   The number of digits in the scaling factor is just sufficient to contain the value of q without leading zeros.

   The length of the intermediate string is:

   ```
   p+3+k
   ```

   where k is the number of digits necessary to represent the value of q (not including a sign or the letter F). For example, given:

   ```
   DECLARE A FIXED(4,-3),
           C CHAR(10);
           A=1234.0E3;
           C=A;
   ```

   The intermediate string generated in converting A would be:

   ```
   b1234F+3
   ```

   which, when assigned to C, would give:

   ```
   b1234F+3bb
   ```

<u>Coded Real Fixed-Point Binary Data:</u> The data item is converted to fixed-point decimal and is transmitted as coded real fixed-point decimal data.

<u>Coded Real Floating-Point Decimal Data:</u> The data item is converted according to the rules for floating-point format items, E(w, d, s). For E-conversion, w = p + n+4, d = p - 1 and s = p.

<u>Coded Real Floating-Point Binary Data:</u> The data item is converted to floating-point decimal with a precision (p) and transmitted as coded real floating-point decimal data.

<u>Coded Complex Data:</u> The data is externally represented as two immediately adjacent real data fields, the left hand field being the real part of the data and the right-hand field being the imaginary part of the data.

A sign always precedes the imaginary part. If the value of the imaginary part is greater than, or equal to, zero, the sign is plus; if the value of the imaginary part is less than zero, the sign is minus. The imaginary part is always followed by the letter I. The field width of the external representation is 2w + 1, where w is as defined above for fixed-point or floating-point output.

<u>NUMERIC CHARACTER DATA:</u> The base of numeric character data may be decimal or binary.

<u>Numeric Decimal Data:</u> The external format and field width of the numeric decimal data item is that described by the associated picture specification.

<u>Numeric Binary Data:</u> The external format and field width of the numeric binary data item is that described by the associated picture specification. The binary digits 0 and 1 are represented by the characters 0 and 1.

<u>Complex Numeric Data:</u> The real and imaginary parts are transmitted as above and the external representation is the concatenation of the real and imaginary parts. The field width is 2w, where <u>w</u> is the number of character positions (or bits, if binary) allocated to the real part of the numeric data; no I is appended.

<u>CHARACTER-STRING DATA:</u> The contents of the character string are written out. If the file has the attribute PRINT, enclosing quotation marks are not supplied, and contained quotation marks are unmodified. The field width is the current length of the string. If the file does not have the attribute PRINT, enclosing quotation marks

are supplied, and contained quotation marks are replaced by two quotation marks. The field width is the current length of the string plus the number of added quotation marks.

BIT-STRING DATA: The format of the data on the external medium is that of a bit-string constant, that is, the value is enclosed in quotation marks and followed by the letter B. The binary bits are represented by the characters 0 and 1. The field width is p+3, where p is the current length of the string, and the three additional positions are for the two quotation marks and the letter B.

Examples of list-directed data specifications:

1. LIST (CARD.RATE, DYNAMIC_FLOW)

2. LIST ((THICKNESS (DISTANCE) DO DISTANCE = 1 TO 1000))

3. LIST (P,Z,M,R)

4. LIST (A*B/C, (X+Y)**2)

The specification in example 4 may only be used for output.

DATA-DIRECTED DATA SPECIFICATION

General format:

Option 1

```
DATA
```

Option 2

```
DATA (data-list)
```

General rules:

1. The data list is described in "Data Lists," in this chapter. It cannot include parameters, or based or defined variables. Names of structure elements need only have enough qualification to resolve any ambiguity; full qualification is not required.

2. On input, option 1 implies that all of the data items to be transmitted are known to the block containing the GET statement; the NAME condition will be raised if a name that is not known to the block is transmitted. On output, it specifies that all data items known to the block and allowed in data-directed transmission are to be transmitted.

3. Recognition of a semicolon in the stream on input causes transmission to cease. On output a semicolon is written into the stream after the last data item transmitted.

Data-Directed Data in the Stream

The data in the stream associated with data-directed transmission is in the form of a list of scalar assignments having the following general format:

```
scalar-variable = constant
[{b|,} scalar-variable = constant]...;
```

General rules:

1. The "scalar variable" may be a subscripted name with decimal integer constant subscripts.

2. On input, the scalar assignments may be separated by either a blank (b in the above format) or a comma. On output, the assignments are separated by blanks.

3. The constant in the general format above has one of the forms as described under "List-Directed Input Format" in this chapter.

General rules for data-directed input:

1. If the data specification in option 1 is used, the names in the stream may be any fully qualified name known at the point of transmission.

2. If option 2 is used, each element of the data list must be an unsubscripted scalar, array, or structure name. The names in the stream must appear in the data list; however, the order of the names need not be the same and the data list may include names that do not appear in the stream. If a name appears in the stream but not in the data list, the NAME condition will be raised.

   For example, consider the following data list, where A, B, C, and D are names of scalar variables:

   ```
   DATA (B, A, C, D)
   ```

   This data list may be associated with the following input data stream:

   ```
   A=2.5, B=.00476, D=125, Z='ABC';
   ```

Note that C appears in the data list but not in the stream and that Z, not in the data list, will raise the NAME condition.

3. If the data list in Option 2 includes the name of an array, subscripted references to that array may appear in the stream. The entire array need not appear.

   Let X be the name of a two dimensional array declared as follows:

```
DECLARE X (2, 3);
```

   Consider the following data list and input data stream:

Data List	Input Data Stream
DATA (X)	X(1,1) = 7.95, X(1,2) = 8085, X(1,3) = 73;

   Although the data list has only the name of the array, the associated input stream may contain values for individual elements of the array.

4. If the data list includes the names of structure elements, then fully qualified names of the items must appear in the stream. Consider the following structures:

```
DECLARE 1 CARDIN, 2 PARTNO, 2 DESCRP,
          2 PRICE,
        1 CARDOUT, 2 PARTNO, 2 DESCRP,
          2 PRICE;
```

   If it is desired to read a value for CARDIN.PARTNO, then the data list and input data stream have the following forms:

Data List	Input Data Stream
DATA (CARDIN.PARTNO)	CARDIN.PARTNO = 737314;

5. Interleaved subscripts cannot appear in qualified names in the stream. All subscripts must be moved all the way to the right, following the last name of the qualified name. For example, assume that Y is declared as follows:

```
DECLARE 1 Y(5,5), 2 A(10), 3 B, 3 C,
        3 D;
```

   An element name would have to appear in the stream as follows:

```
        Y.A.B(2,3,8)=8.72
```

   The name in the data list, of course, could not contain the subscript.

General rules for data-directed <u>output</u>:

1. An element of the data list, which can be subscripted may be a scalar variable, an array variable, a structure variable, a repetitive specification involving any of these elements or further repetitive specifications. The data with names appearing in the data list is transmitted in the form of a list of scalar assignments separated by blanks and terminated by a semicolon. Tabs and line splitting for PRINT file data items follow the rules set for list-directed transmission.

2. Array variables in the data list are treated as a list of the contained subscripted elements in row-major order.

   Let X be an array declared as follows:

```
DECLARE X (2,4);
```

   Let X appear in a data list as follows:

```
DATA (X)
```

   Then, on output, the output data stream is as follows:

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

3. Items that are part of a structure appearing in the data list are transmitted with the full qualification, but subscripts follow the qualified names rather than being interleaved. If a data list is specified for a structure element transmitted under data-directed output as follows:

```
DATA (Y(1,3).Q)
```

   then the associated data field in the output stream is as follows:

```
Y.Q(1,3) = 3.756;
```

4. Structure names in the data list are interpreted as a list of the contained scalar or array elements, and arrays are treated as above.

   Consider the following structure:

```
1 A, 2 B, 2 C, 3 D
```

   If a data list for data-directed output is as follows:

```
DATA (A)
```

```
AB:  PROCEDURE;
                                            Input Stream
     DECLARE A(6), B(7);
                                            B(1)=1, B(2)=2, B(3)=3,
     GET FILE (X) DATA (B);
                                            B(4)=1, B(5)=2, B(6)=3, B(7)=4;
     DO I = 1 TO 6;

     A (I) = B (I+1) + B (I);
                                            Output Stream
     END;
                                            A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
     PUT FILE (Y) DATA (A);
                                            A(5)= 5 A(6)= 7;
     END AB;
```

Figure 3. Example of Data-Directed Transmission, both Input and Output

then, if the values of B and D were 2 and 17 respectively, the associated data fields in the output stream would be as follows:

```
A.B= 2 A.C.D= 17;
```

Length of Data-Directed Data Fields

The length of the data field on the external medium is a function of the internal precision, the value of the data item being written, and the length of the data identifier and its associated subscript list. The field length for coded arithmetic data, numeric field data, and bit-string data is the same as described for list-directed output (see "Format of List-Directed Output Fields"). Subscripts are printed as possibly signed decimal integer constants with no leading blanks.

For character-string data, the contents of the character string are written out enclosed in quotation marks. Each quotation mark contained within the character string is represented by two successive quotation marks.

Example:

Assume that A is declared as a one-dimensional array of six elements; B is a one-dimensional array of seven elements. The procedure in Figure 3 calculates and writes out values for A(I) = B(I+1) + B(I).

EDIT-DIRECTED DATA SPECIFICATION

General format:

```
EDIT (data-list)(format-list)
   [(data-list)(format-list)]...
```

General rules:

1. The data list general rules are given in "Data Lists," and the format list general rules in "Format Lists." This form of transmission can be used for sterling values.

2. On output, the value of each data item in the data list is converted to a format specified by the associated format item in the format list. The first scalar data item is associated with the first format item. If the format item is a control format item, the control item is executed, and the data item associated with the first name in the data list is then associated with the next format item. The second scalar data item is then associated with the second data format item, etc. Suppose the format list specifies <u>j</u> data format items, and the data list specifies <u>k</u> data items. Then, if j<k, after <u>j</u> scalar data items have been transmitted, the format list is re-used, the (j+1)th scalar item being associated with the first format item, etc. This re-use is performed as many times as required. If j>k, excessive format items are ignored.

3. For input, data in the stream is considered to be a continuous string of characters not separated into individual data items. The number of characters for each data item is specified by a format item in the format list. The characters are treated

according to the associated format item.

4. An array or a structure in a data list is equivalent to n data items, where n is the number of scalar elements in the array or structure.

5. The specified transmission is complete when the last data item has been processed using its corresponding format item. Subsequent format items, including control format items, are ignored.

Examples:

The first of the following examples is an edit-directed input specification, and the second is an output specification.

```
1. EDIT     (NAME,     DATE,     SALARY)
   (A(COLA-COLB), X(2), A(6), F(M +2,2))

2. EDIT  ('INVENTORY-'  ||  INUM,INVCODE)
   (A, F(5))
```

FORMAT LISTS

The edit-directed data specification requires an associated format list.

General format of a format list:

```
        (format-list)
```

where "format list" is defined as:

```
⎧item            ⎫ ⎡, item            ⎤
⎨n item          ⎬ ⎢, n item          ⎥...
⎩n(format-list)  ⎭ ⎣, n(format-list)  ⎦
```

Syntax rules:

1. Each "item" represents a format item as described below.

2. The letter n represents an iteration factor, which is either an expression enclosed in parentheses, or a decimal integer constant. If a decimal integer constant is used, at least one blank must must follow it. The iteration factor specifies that the associated format item is to be used n successive times. A zero or negative iteration factor specifies that the associated format item is to be skipped and not used (the data list item will be associated with the next format item). If an expression is used to represent the iteration factor, it is evaluated and converted to an integer once for each set of iterations. The associated format item is that item or list of items to the right of the iteration factor.

3. A format list always must be delimited by parentheses.

General rule:

There are two types of format items: data format items and control format items. Data format items specify the form of data fields in the stream. Control format items specify page, line, and spacing operations.

Data Format Items

Data format items describe data representation in the data stream.

The discussion of format items requires the following definitions:

1. The letter w represents the length of the data field, in characters, used by the external representation (including signs, decimal points, blanks, and the letter E as used in the representation of constants).

2. The letter d represents the number of positions after the decimal point.

3. The letter s represents the number of significant digits to appear.

4. The letter p represents a scaling factor, which may be positive or negative.

Any of the quantities w, d, s, and p may be specified by a scalar expression. When the format item is used, the expression is evaluated and converted to an integer. If $w \le 0$ in a format specification, then the associated data and format list items are skipped, unless, on input, w=0 and the data item is a string, in which case, the data value is taken as the null string. On output, the format list item is skipped if w is less than or equal to zero. The quantity d must be less than or equal to s, and s must be less than or equal to w.

On input, the data item in the external data field is treated as if it conformed to the characteristics described by the format item.

There are six format items associated with data: fixed-point (F), floating-point (E), complex (C), picture specification (P), character string (A), and bit string (B).

FIXED-POINT FORMAT ITEMS: Decimal numeric data may be described by a fixed-point format item.

General format:

Option 1
```
    F(w)
```

Option 2
```
    F(w,d)
```

Option 3
```
    F(w, d, p)
```

General rules:

1. On input, the data item in the external data field is the character representation of a decimal fixed-point number anywhere in a field of width w. Leading and trailing blanks are ignored, but if the data consists only of blanks, it is interpreted as zero.

   In option 2, if no decimal point appears in the number, it is assumed to appear immediately before the last d digits (trailing blanks are ignored). If a decimal point does appear, it overrides the d specification. Option 1 is treated as Option 2, with d equal to zero.

   In Option 3, the scaling factor effectively multiplies the external data value by 10 raised to the value of p. If p is positive, the number is treated as though the decimal point appeared p places to the right of its given position. If p is negative, the data is treated as though the decimal point appeared p places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it is given, or by d, in the absence of an actual point.

2. On output, the external data is a decimal fixed-point number, right-adjusted in a field of width w. If the right-adjustment results in low-order digits being removed, the remaining lowest-order digit is rounded if it was followed by a digit greater than or equal to 5.

   In Option 1, only the integer portion of the number is written; no decimal point appears.

   In Option 2, both the integer and fractional parts of the number are written. If d is greater than 0, a decimal point is inserted before the last d digits, and the value is appropriately positioned. Trailing zeros are supplied if the number of fractional digits is less than d (where d must be less than w). If the absolute value is less than 1, a zero precedes the point; if w is not large enough to include the zero, the SIZE condition wll be raised.

   In Option 3, the scaling factor effectively multiplies the internal data value by ten raised to the power of p, before it is edited into its external character representation. If d is zero, only the integer portion of the number is considered.

   For all options, if the value of the data item is less than zero, a minus sign will be prefixed to the external character representation; if it is greater than or equal to zero, no sign will appear. Therefore, for negative values, w must encompass both sign and decimal point. If the length of the data item is greater than w, the SIZE condition is raised.

FLOATING-POINT FORMAT ITEMS: Decimal numeric data may be described by a floating-point format item.

General format:

```
E(w, d[, s])
```

General rules:

1. On input, the data item in the external data field is an optionally signed character representation of a decimal floating-point number anywhere within a field of width w. An all-blank field is not treated as zero; it causes the CONVERSION condition to be raised. The mantissa is a fixed decimal constant.

   The external form of the number is as follows:

$$[\pm]\ \text{mantissa}\ \left[\begin{Bmatrix}[\text{E}]\ \pm \\ \text{E}\ [\pm]\end{Bmatrix}\ \text{exponent}\right]$$

   a. If there is no decimal point in the data field, the decimal point is assumed to be before the last d digits of the mantissa. If there is a decimal point in the data field, it overrides the decimal point placement specified by d. Note that trailing blanks in the data field are ignored.

   b. The "exponent" is a decimal integer constant. If the exponent and the preceding E or sign are omitted, a zero exponent is assumed.

2. On output, the data item in the data field has the following general form:

   [-] s-d digits.d digits E{±} exponent

   a. The "exponent" is a decimal integer constant of n digits, where n is defined individually for each implementation. The exponent is adjusted so that the leading digit of the mantissa is nonzero. Unless the value of the data is zero, at least one non-fractional significant digit always will appear. In the case of the value zero, one zero digit appears before the point and d zero digits after the point; the exponent is also zero.

   b. If the above form does not fill the field of width w, it is right-adjusted. If the right-adjustment results in low-order digits being removed, the remaining lowest-order digit is rounded if it was followed by a digit greater than or equal to 5. If s is omitted it is taken as equal to d + 1. The field width w must be greater than or equal to s + n + 3 for non-negative values, and s + n + 4 for negative values of the data item. However, if d is zero, the decimal point is not written, and w is equal to s+n+2. If the length of the data item is greater than w, the SIZE condition is raised.

COMPLEX FORMAT ITEMS: Complex data may be described by a complex format item.

General format:

```
C(real-format-item
   [, real-format-item])
```

General rules:

1. Each "real format item" is specified by F, E, or P formats. P can specify a numeric field only; it cannot specify a sterling picture.

2. On input, the external data is the real and imaginary parts of the complex number in adjacent fields described by the two contained format items. If the second real format item is omitted, it is assumed to be the same as the first.

3. On output, the form of the real and imaginary parts is specified by the real format items. If the second is omitted, it is assumed to be the same as the first.

PICTURE FORMAT ITEM: Numeric data may be described by a numeric picture using the P format item. The picture format item allows transmission of sterling data items.

General format:

```
P 'numeric-picture-specification'
```

The "numeric picture specification" is described in "The PICTURE Attribute," in Chapter 4.

On input, the picture specification describes the form of the data on the external medium and how it is to be interpreted numerically.

On output, the value of the list item is edited to the form specified by the picture before it is transmitted.

BIT-STRING FORMAT ITEMS: The bit-string item describes the data field representation of a bit string using the characters 0 and 1.

General format:

```
B   (w)
```

General rules:

1. In the case of input, w is always required. For output, if w is omitted, it is taken to be the current length of the associated bit-string data-list element; w must be specified if conversion is to be performed.

2. On input, the data field is a character representation of bit string anywhere within the field of width w. If the data field contains only blanks, or any characters other than zero or one, the CONVERSION condition is raised.

3. On output, the character representation of the bit string is left-adjusted in the field of width w. Truncation, if necessary, is performed on the right. Blanks are used for padding.

CHARACTER-STRING FORMAT ITEMS: Character data may be described by a character-string format item.

General format:

```
{A   (w)                               }
{P   'character-picture-specification' }
```

General rules:

1. The external representation is a string of w characters.

2. On input, truncation, if necessary, is performed on the right. If the associated list element is too short, it is extended on the right with blanks. If the picture form is used, w is implied. Checking is performed. On input, w is always required.

3. On output, w can be omitted, in which case w is taken to be the current length of the string (or the length of the converted character string).

Control Format Items

There are three types of control format items, the spacing format item X, the positioning format items SKIP and COLUMN, and the printing format items PAGE and LINE.

Spacing Format Item

The spacing format item specifies relative horizontal spacing.

General format:

```
X (w)
```

General rules:

1. On input, the format item specifies that the next w characters of the stream are to be ignored.

2. On output, the format item specifies that w blank characters are to be inserted into the stream.

3. If w is less than zero, it is taken as zero.

Positioning Format Items

The positioning format items specify positioning to a new current line or to a specified column in the current (or next) line. (The length of a line is derived from the linesize of the file.)

General format:

```
SKIP [(w)]
COLUMN (w)
```

General rules:

1. The SKIP format item operates in the same manner as the SKIP option of a GET or PUT statement.

2. The COLUMN format item specifies that the file is to be positioned to the wth column of the current line. If the file is an output file, blank characters are inserted into the stream until the wth column of the line is reached. If the file is an input file, characters are ignored until the wth column is reached. If the file is already positioned beyond the wth column of the current line, the data set is positioned to the wth column of the next line. If w is less than 1 or greater than the linesize of the file, w is assumed to be 1.

Printing Format Items

The printing format items can be used only with STREAM PRINT files.

General format:

```
PAGE
LINE (w)
```

General rule:

The PAGE and LINE format items operate in the same manner as the corresponding options with the PUT statement.

Note that X and COLUMN specify, respectively, relative horizontal spacing and absolute horizontal spacing. Similarly, SKIP and LINE specify relative vertical positioning and absolute vertical positioning. The first line on any page is line number one.

Remote Format Item

If it is desired to locate format items remotely from a format list, the remote format item, R, may be used.

General format:

```
R(statement-label-designator)
```

General rules:

1. The "statement label designator" is a label constant or a label variable that has as its value the statement label of a FORMAT statement. The FORMAT statement includes a format list that is taken to replace the format item.

2. The R format item and the specified FORMAT statement must be internal to the same invocation of the same block.

3. There can be no recursion. That is, a remote FORMAT statement may not contain an R format item which names itself as a statement label designa-

tor, nor may it name another remote FORMAT statement that will lead to the naming of the original FORMAT statement through a statement label designator. This is assured if the FORMAT statement referred to by a remote format item does not itself contain a further remote format item.

4. Any conditions enabled for the GET or PUT statement must be correspondingly enabled for the remote FORMAT statements utilized.

5. If the GET or PUT statement is the single statement of an on-unit, it cannot contain a remote format item.

6. A FORMAT statement encountered in sequential flow of control is ignored.

DATA STREAM TRANSMISSION STATEMENTS

This section provides a summary of the allowed STREAM transmission statements, along with their options, according to file attributes (the statements are discussed individually in Chapter 8).

STREAM INPUT:

```
    ┌                                          ┐
    │FILE    (filename) [COPY]                 │
GET │        [SKIP[(scalar-expression)]]       │
    │STRING  (scalar-character-string-         │
    │        variable)                         │
    └                                          ┘
             [data-specification];
```

STREAM OUTPUT:

```
    ┌                                          ┐
    │FILE    (filename)                        │
PUT │        [SKIP[(scalar-expression)]]       │
    │STRING  (scalar-character-string-         │
    │        variable)                         │
    └                                          ┘
             [data-specification];
```

STREAM OUTPUT PRINT:

```
PUT   [FILE  (filename)]
             [data-specification]
            ┌PAGE [LINE (expression)] ┐
            │SKIP [(expression)]      │
            └LINE (expression)        ┘
```

Note: The "data specification" can be omitted only if the SKIP option or one of the printing options appears.

The data specification can have one of the following forms:

```
LIST (data-list)
DATA [(data-list)]
EDIT(data-list)(format-list)
    [(data-list)(format-list)]...
```

Data lists and format lists are discussed earlier in this chapter. Format lists may use any of the following format items:

A,B,C,E,F,P,R,X, SKIP,COLUMN	which may be used with any STREAM file
PAGE,LINE	which may be used only with STREAM OUTPUT PRINT files
A,B,C,E,F,P,R,X	which may be used with the STRING option

RECORD TRANSMISSION

Data sets that contain discrete records or which are to be created as a collection of discrete records may be manipulated with record operation statements. The record operation statements are READ, WRITE, REWRITE, LOCATE, DELETE, and UNLOCK. A general description of these statements is contained in this chapter, and they are described completely in Chapter 8. The records obtained from data sets or dispatched to data sets are defined in terms of the data attributes of a variable. For input operations the record is obtained from the data set and placed intact into the variable. For output operations, the variable is transmitted intact into the data set.

The variables involved in record transmission must be unsubscripted, of level 1 (scalar variables and array variables are of level 1 by default), and of any storage class. The variables cannot be parameters or defined variables. They may contain VARYING length strings. They may contain LABEL, EVENT, TASK, and POINTER variables, but such data may lose its validity in transmission. OFFSET variables, however, will maintain their validity.

With RECORD transmission, it is possible to operate upon the record in a buffer if the file has the BUFFERED attribute. Operations within the buffer are accomplished through the use of a based variable, which describes the data attributes of the record, and a pointer variable, which identifies the location of the record within the buffer. Note that an offset variable cannot be used, since an offset value is relative only to its associated area variable.

For input/output operations specifying based variables, the pointer value is set by the SET option in the READ or LOCATE statements.

RECORD TRANSMISSION STATEMENTS

This section provides a summary of the allowed RECORD transmission statements, along with their options, according to file attributes (the statements are discussed individually in Chapter 8).

<u>SEQUENTIAL BUFFERED INPUT:</u>

```
READ  FILE (filename)
    INTO (variable) [KEYTO
     (character-string-variable)];

READ  FILE (filename)
    SET (pointer-variable)
    [KEYTO
     (character-string-variable)];

READ  FILE (filename)
    [IGNORE (expression)];

READ  FILE (filename)
    INTO (variable)
    KEY (expression);

READ  FILE (filename)
    SET (pointer-variable)
    KEY (expression);
```

<u>SEQUENTIAL BUFFERED OUTPUT:</u>

```
WRITE FILE (filename)
    FROM (variable)
    [KEYFROM (expression)];

LOCATE   variable   FILE (filename)
    [SET        (pointer-variable)]
    [KEYFROM (expression)];
```

<u>SEQUENTIAL BUFFERED UPDATE:</u>

```
READ  FILE (filename)
    INTO (variable)
    [KEYTO
     (character-string-variable)];

READ  FILE (filename)
    SET (pointer-variable)
    [KEYTO
     (character-string-variable)];

REWRITE FILE (filename);

REWRITE FILE (filename)
    FROM (variable);

READ  FILE (filename)
    [IGNORE(expression)];

READ  FILE (filename)
    INTO (variable)
    KEY (expression);

READ  FILE (filename)
    SET (pointer-variable)
    KEY (expression);

DELETE  FILE(filename);
```

<u>SEQUENTIAL UNBUFFERED INPUT:</u>

```
READ  FILE (filename)
    INTO (variable)
    [KEYTO
     (character-string-variable)]
    [EVENT (event-variable)];

READ  FILE (filename)
    [IGNORE (expression)]
    [EVENT (event-variable)];

READ  FILE (filename)
    INTO (variable
    KEY (expression)
    [EVENT (event-variable)];
```

<u>SEQUENTIAL UNBUFFERED OUTPUT:</u>

```
WRITE FILE (filename)
    FROM (variable)
    [KEYFROM (expression)]
    [EVENT (event-variable)];
```

<u>SEQUENTIAL UNBUFFERED UPDATE:</u>

```
READ  FILE (filename)
    INTO (variable)
    [KEYTO
     (character-string-variable)]
    [EVENT (event-variable)];

REWRITE  FILE (filename)
    FROM (variable)
    [EVENT (event-variable)];

READ  FILE (filename)
    [IGNORE (expression)]
    [EVENT (event-variable)];

READ  FILE (filename)
    INTO (variable)
    KEY (expression)
    [EVENT (event-variable)];

DELETE  FILE(filename)
    [EVENT (event-variable)];
```

<u>DIRECT INPUT:</u>

```
READ  FILE (filename)
    INTO (variable)
    KEY (expression)
    [EVENT (event-variable)];
```

DIRECT OUTPUT:

```
WRITE FILE (filename)
    FROM (variable)
    KEYFROM (expression)
    [EVENT (event-variable)];
```

DIRECT UPDATE:

```
READ  FILE (filename)
    INTO (variable)
    KEY (expression)
    [EVENT (event-variable)];

REWRITE  FILE (filename)
    FROM (variable)
    KEY (expression)
    [EVENT (event-variable)];

WRITE FILE (filename)
    FROM (variable)
    KEYFROM (expression)
    [EVENT (event-variable)];

DELETE  FILE (filename)
    KEY (expression)
    [EVENT (event-variable)];
```

DIRECT UPDATE EXCLUSIVE:

```
READ  FILE (filename)
    INTO (variable)
    KEY (expression)  [NOLOCK]
    [EVENT (event-variable)];

REWRITE  FILE (filename)
    FROM (variable)
    KEY (expression)
    [EVENT (event-variable)];

WRITE FILE (filename)
    FROM (variable)
    KEYFROM (expression)
    [EVENT (event-variable)];

DELETE  FILE (filename)
    KEY (expression)
    [EVENT (event-variable)];

UNLOCK  FILE (filename)
    KEY (expression);
```

RECORD TRANSMISSION OPERATIONS

1. A SEQUENTIAL file specifies that the accessing, creation, or modification of the data set records is performed in a particular order, that is, from the first record of the data set to the last record of the data set (or, if the BACKWARDS attribute is specified, from the last to the first).

2. A DIRECT file specifies that the accessing, creation, or modification of the data set records is performed by indicating which particular record of the data set is to be operated upon.

3. A data set that is accessed, created, or modified in the SEQUENTIAL access method may or may not have the attribute KEYED. If a data set has been created with the KEYED attribute, any recorded keys actually present in the data set may be ignored while accessing sequentially, or they may be extracted from the data set by use of the KEYTO option. It is possible to create a KEYED data set as a SEQUENTIAL OUTPUT file and later to access that data set as a DIRECT file.

4. SEQUENTIAL INPUT and SEQUENTIAL UPDATE files may be positioned to a particular record within the data set by a READ operation that specifies the key of the desired record. Thereafter, successive READ statements without the KEY option will access the records sequentially. This kind of accessing may be used only if the data set contains keyed records and if the file has the KEYED attribute.

5. Existing records of a data set in a SEQUENTIAL UPDATE file can be rewritten (REWRITE statement), ignored (READ statement with an IGNORE option), or deleted (DELETE statement), but the number of records cannot be increased. Note that when deleting a record from a SEQUENTIAL UPDATE file, the programmer cannot explicitly identify the record to be deleted; only the last record that was read can be deleted. On the other hand, for DIRECT UPDATE files, the programmer can and must explicitly identify the record to be deleted by the DELETE statement. In addition, he can add records to a DIRECT UPDATE file as well as rewrite them by using WRITE and REWRITE statements, respectively.

6. If the READ INTO option is used in referring to a SEQUENTIAL BUFFERED UPDATE file and the next REWRITE statement does not make use of a FROM option, the record in the data set is replaced from the buffer and not from the variable that had been specified in the INTO option of the READ statement. The FROM option in a REWRITE statement must specifically name the variable INTO which the data has been read if that data is to be rewritten.

7. Operations upon a data set accessed sequentially may lead to erroneous

results if the same data set or file is being referred to asynchronously in more than one task. The separate tasks might use different filenames, but if the different file openings identify the same data set, the tasks would refer to the same set of records.

8. A data set being accessed directly is suitable for asynchronous operations because the reference to the data set does not imply any explicit ordering of the records and because the records are transmitted INTO and FROM variables that can be known only within the individual tasks. This is true whether the data set is identified by more than one file opening or is referred to through use of the same filename.

9. When a file has the DIRECT UPDATE EXCLUSIVE attributes, it is possible to protect individual records from simultaneous updating by differant tasks. For an EXCLUSIVE file, any READ statement without a NOLOCK option automatically locks the record read. No other task operating upon the same file can access a locked record until it is unlocked by the locking task. Any task (other than the locking task) referring to a locked record will wait at that point until the record is unlocked. A record can be explicitly unlocked by the locking task through execution of a REWRITE, DELETE, or UNLOCK statement for the same record. Records are unlocked automatically when the file is closed or upon completion of the locking task. The EXCLUSIVE attribute applies only to the file and not to the data set. Consequently, record protection is provided only if all tasks refer to the data set through use of the same file; if they refer to the same data set using different files, the protection does not apply. To ensure protection, the data set to which reference is made by more than one task through the same file must be opened by a parent of all these tasks. Note that a reference to a file parameter and a reference to its associate argument are references to the same file.

10. A WRITE statement adds records to a data set, while a REWRITE statement replaces records. Thus, a WRITE statement may be used with OUTPUT or UPDATE files, while a REWRITE statement may only be used with UPDATE files. Moreover, a WRITE statement may use the KEYFROM option to indicate the actual transference of a key from internal storage to the data set; the REWRITE statement uses the KEY option to identify the existent record to be replaced.

SYSIN AND SYSPRINT

A GET statement that does not specify a file or string option is equivalent to the GET statement:

```
GET FILE(SYSIN)...;
```

A PUT statement that does not specify a file or string option is equivalent to the PUT statement:

```
PUT FILE(SYSPRINT)...;
```

The contextual recognition of the FILE attribute applies to the identifiers SYSIN and SYSPRINT in these statements.

If the merged attributes of a file named SYSPRINT contain the attributes STREAM and OUTPUT and if SYSPRINT is not internal, the default attribute of PRINT is supplied.

This section includes a description of each statement in the language. These descriptions are presented in alphabetic order.

To show the relationships among these statements, they are also classified into logical groups.

RELATIONSHIP OF STATEMENTS

CLASSIFICATION

Statements may be classified into the following logical groups: assignment, control, declaration, error control and debug, input/output, program structure, and storage allocation.

Assignment Statement

The assignment statement is used to evaluate expressions and to assign values to scalars, arrays, and structures.

Control Statements

The control statements affect the normal sequential flow of control through a program. The control statements are GO TO, IF, DO, CALL, RETURN, WAIT, STOP, EXIT, and DELAY.

Data Declaration Statement

The data declaration statement, DECLARE, specifies attributes for identifiers. This statement is described in Chapter 4.

Error Control and Debug Statements

When an interrupt occurs during program execution, standard operating system action is taken; however, the language provides the facility to override system action on these interrupts. By using the ON statement, a programmer may specify the action to be taken when an interrupt occurs and can record the status of the program at the point of the interrupt. By using the SIGNAL statement, the programmer may initiate programmed interrupts and may simulate machine interrupts to facilitate debugging.

Input/Output Statements

The input/output statements may be classified as follows: file preparation, record status, data specification, and data transmission.

File Preparation Statements

The OPEN statement associates a filename with a data set and completes the specification of the attributes of the file, in preparation for input/output on a file. The CLOSE statement dissociates the filename from the data set and thereby releases the filename for use in connection with any other data set.

Record Status Statements

The DELETE statement deletes a record from an UPDATE file. The UNLOCK statement makes accessible a record which would otherwise be inaccessible as a result of the READ statement accessing from an EXCLUSIVE file.

Data Specification Statements

The format of data fields to be transmitted may be specified by the FORMAT statement or in the GET or PUT data transmission statements.

Data Transmission Statements

The GET and PUT statements cause values to be transmitted between a data set and specified variables in the program. The READ and WRITE statements cause a single record to be transmitted between a data set and variables in the program. The REWRITE statement specifies the updating of an existing record of the data set. The LOCATE statement permits a record to be created in the buffer storage and subsequently written. The DISPLAY statement causes messages to be transmitted between the program and the machine operator.

Program Structure Statements

The program structure statements are: PROCEDURE, BEGIN, END, DO, and ENTRY. The first three statements delimit the scope of declarations within a program. The ENTRY statement provides a secondary entry point for a procedure.

Storage Allocation Statements

The storage allocation statements are ALLOCATE and FREE. These statements allocate and free storage for variables.

SEQUENCE OF CONTROL

Within a block, control normally passes sequentially from one statement to the next. If a DECLARE, FORMAT, or ENTRY is encountered, control passes to the next statement. If an internal PROCEDURE statement is encountered, control passes to the statement following the end of the procedure. Control passes to the statement following an IF statement when control reaches the end of the THEN-unit. Sequential operation is also modified by the following statements: CALL, DO, END, EXIT, GO TO, PROCEDURE, RETURN, SIGNAL, and STOP.

A CALL statement passes control to the specified entry point.

A DO statement defines a group that is treated as a single statement and can cause repeated execution of a group.

An END statement, logically terminating a procedure, acts as a RETURN statement, causing control to return to the invoking procedure.

The EXIT statement causes control to leave a task; the STOP statement causes control to leave a program.

A GO TO statement causes control to transfer to the specified statement label.

A PROCEDURE statement heads a procedure. Procedures may be considered as independent blocks and are placed anywhere within an external procedure, consistent with desired identifier scopes. However, a procedure may be invoked only by a CALL statement, a statement with a CALL option, or a function reference. Thus, control passes around a nested procedure, from the statement before a PROCEDURE statement to the statement after the appropriate END statement for the procedure.

A RETURN statement returns control from a procedure to the invoking procedure.

A SIGNAL statement specifying an enabled condition causes control to pass to the on-unit of the associated ON statement. If there is no associated ON statement, control is passed to the appropriate system routine.

The following conditions may cause sequential operation to be modified:

1. A function reference in any expression causes control to pass to the specified function procedure.
2. The occurrence of an enabled condition specified in an ON statement causes control to pass to the associated ON-unit. If there is no ON statement, control is passed to the appropriate system routine.
3. The flow of control through the IF and ON statements and through a DO group may or may not be sequential.
4. In an appropriate environment, the asynchronous execution of several operations may involve transfer of control under the influence of external occurrences.

The following example illustrates sequence of control:

```
A:  PROCEDURE;
B:  X = Y + Z;
C:  CALL D;
E:  W = P*Q;
    D:  PROCEDURE;
    G:  S = T/P;
    H:  RETURN;
    I:  END D;
J:  U = V**W;
K:  GO TO N;
    .
    .
    .
N:  END;
```

Control flows in the following order: A, B, C, D, G, H, E, J, K, N.

ALPHABETIC LIST OF STATEMENTS

The ALLOCATE Statement

Function:

The ALLOCATE statement causes storage to be allocated for specified controlled and/or based variables.

General format:

Option 1:

```
ALLOCATE [level] identifier
   [dimension] [attribute]...
   [,[level] identifier [dimension]
   [attribute]...]...;
```

Option 2:

```
ALLOCATE based-variable-identifier
   [SET (scalar locator-variable)]
   [IN (scalar area-variable)]
   [, based-variable-identifier
   [SET (scalar locator-variable)]
   [IN (scalar area-variable)]]...;
```

Syntax rules:

1. Based variables and controlled variables may both be allocated in the same ALLOCATE statement.

Syntax rules 2 through 6 apply only to Option 1:

2. Each identifier must represent data of the controlled storage class or be an element of a controlled major structure.

3. "Dimension" indicates a dimension attribute. "Attribute" indicates an AREA, BIT, CHARACTER, or INITIAL attribute. "Level" indicates a level number.

4. A dimension attribute, if present, must specify the same number of dimensions as that declared for the associated identifier.

5. The attribute BIT may appear only with a BIT identifier; CHARACTER may appear only with a CHARACTER identifier; AREA may appear only with an AREA identifier.

6. A structure element name, other than the major structure name, may appear only if the relative structuring of the entire structure appears as in the DECLARE statement for that structure. In this case, dimension attributes must be specified for all identifiers that are declared with the dimension attribute.

Syntax rules 7 and 8 apply only to Option 2:

7. The based variable appearing in the ALLOCATE statement may be a scalar variable, an array, or a major structure. When it is a major structure, only the major structure name is specified.

8. The SET clause, if present, may appear preceding or following the IN clause. The SET clause must appear unless a locator variable has been specified in the BASED attribute declaration for the variable, in which case it is optional.

General Rules:

Rules 1 through 5 apply only to Option 1:

1. When Option 1 is used, an ALLOCATE statement for an identifier for which storage was allocated and not freed causes storage for the identifier to be "pushed down" or stacked. This pushing down creates a new generation of data for the identifier. When storage for this identifier is freed, using the FREE statement, storage is "popped up" or removed from the stack.

2. Bounds of arrays, lengths of strings and sizes of areas are fixed at the execution of an ALLOCATE statement.

   a. If a bound, length, or size is explicitly specified in an ALLOCATE statement, it overrides the specification given in the DECLARE statement.

   b. If a bound, length, or size is specified by an asterisk in an ALLOCATE statement, that value is taken from the most recent allocation. If the variable has not been previously allocated, the bound, length, or size is undefined.

   c. Either the ALLOCATE statement or the DECLARE statement must specify any necessary dimension, size, or length attributes for an identifier. Any expression taken from the DECLARE statement is evaluated at the point of allocation using the condition enabling of the ALLOCATE statement, although the names are interpreted in the environment of the DECLARE statement.

   d. If, in either an ALLOCATE or a DECLARE statement, bounds, lengths, or area sizes are specified by expressions that contain references to the variable being allocated, the expressions are evaluated using the value of the most recent generation of the variable.

3. Upon allocation of an identifier, initial values are assigned to it if the identifier has an INITIAL attribute in either the ALLOCATE statement or

DECLARE statement. Expressions or a CALL option in the INITIAL attribute are executed at the point of allocation, using the condition enabling the ALLOCATE statement, although the names are interpreted in the environment of the declaration. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, the INITIAL attribute in the ALLOCATE statement is used. If initialization involves reference to the variable being allocated, the reference will be to the new generation of the variable.

4. A parameter that is declared CONTROLLED may be specified in an ALLOCATE statement.

5. The evaluations implied by the ALLOCATE statement are subject to the same interdependency and irreducibility rules as those for the evaluations involved in prologue activity (see "Prologues," in Chapter 6).

Rules 6 through 11 apply only to Option 2:

6. When Option 2 is used, storage is not "pushed down" or stacked. A given generation of a based variable may be accessed by a suitable based reference regardless of allocations of the based variable performed after this generation is allocated. The allocation of a based variable proceeds as follows:

   a. Bounds, string lengths, and area sizes of all the fields are evaluated in an implementation-defined order. Expression preceding the keyword REFER are used as the values of the bounds, string lengths, or area sizes specified by the REFER options.

   b. Sufficient storage for a generation of the based variable with these bounds, string lengths, and area sizes is allocated. This may raise the AREA condition if the allocation is attempted in an area.

   c. Within the newly allocated generation, those variables that are objects of REFER options are initialized to the values specified in the REFER options. This initialization is performed in an implementation-defined order.

   d. The locator variable specified in the SET option or, in its absence, the locator variable specified in the BASED attribute of the based variable declaration, is assigned a pointer value which identifies the generation that has been allocated.

   e. Initial values specified in the declaration of the based variable are assigned to the generation that has been allocated.

<u>Note:</u> Stages c and d may be performed in either order.

7. The allocation of a based variable involves the based variable to be allocated, a locator variable to identify the new generation, and an area if the generation is to be allocated in an area. If no SET option is specified, a SET option is assumed to specify the locator variable given in the BASED attribute of the based variable declaration; it is an error, in such a case, if this BASED attribute does not specify a locator variable. If the SET option specifies an offset variable and no IN option is present then an IN option is assumed to specify the area given in the OFFSET attribute of the offset variable declaration; in such a case, it is an error if this OFFSET attribute does not specify an area variable.

8. If the SET option specifies an offset variable, the pointer value identifying the new generation is assigned to the offset variable; the IN option must be present, or be assumed, and it must specify either the same area as that specified in the OFFSET attribute of the offset variable declaration, or an area contained in or containing that area.

9. If no IN option is present and none is assumed, the new generation is allocated in storage associated with the task which executes the ALLOCATE statement. The SET option in this case must specify a pointer variable.

10. If an IN option is present, or is assumed, an attempt is made to allocate the new generation in the area specified in the IN option. If there is sufficient storage the generation is allocated in the area and a pointer value identifying the generation is assigned to the locator variable specified in the SET option. If insufficient storage exists, the AREA condition is raised. On normal return from an AREA on-unit, the IN option is re-evaluated, and the allocation is attempted again.

11. A pointer value identifying an area does not necessarily compare equal with a pointer value identifying the

first generation allocated within the area.

Examples:

1. The following examples illustrate the use of the ALLOCATE statement for a controlled identifier:

```
DECLARE A(N1,N2) CONTROLLED ;
```

N1, N2 = 10;	
ALLOCATE A;	The bounds are 10 and 10
ALLOCATE A (K1,K2);	The bounds are K1 and K2 which override N1 and N2.
N1 = N1 + 1;	
ALLOCATE A;	The bounds are 11 and 10.
ALLOCATE A (*,*);	The bounds are 11 and 10.
ALLOCATE A (J1, J2);	The bounds are J1 and J2.

2. The following example illustrates the use of the ALLOCATE statement when the DECLARE statement contains asterisks for the length of a controlled bit string B:

```
DECLARE B BIT (*)  VARYING CONTROLLED ;
```

ALLOCATE B BIT (*);	Illegal; violates rule 2b.
ALLOCATE B;	Illegal; violates rule 2c.
ALLOCATE B BIT (N);	The maximum length is N.

3. The following example illustrates the use of the built-in function ALLOCATION and of the INITIAL attribute for a controlled variable in an ALLOCATE statement:

```
DECLARE   A(N,N)   CONTROLLED   INITIAL
((N*N)0);
  .
  .
  .
IF ¬ ALLOCATION  (A)  THEN  ALLOCATE  A
INITIAL (1,(N-1) ((N)0,1));
  .
  .
  .
ALLOCATE A;
```

4. The following example illustrates three uses of Option 2 of the ALLOCATE statement for based identifiers.

```
DECLARE VALUE BASED (P),
    RATES (10) BASED (Q),
    1 GROUP BASED (R),
      2 J FIXED BINARY,
      2 PTS (EXT REFER (J)) POINTER,
      2 VALUES (10) FIXED,
    TABLE AREA STATIC EXTERNAL,
      S POINTER;
```

a. ALLOCATE VALUE SET (P);
Allocates space in systems storage for a generation of the based variable VALUE, and sets the pointer variable P to identify the particular generation.

b. ALLOCATE GROUP SET (R);
Allocates space in systems storage for a generation of the structure GROUP, and sets the pointer variable R to identify the generation. The dimension of each of the component PTS is determined by the value of EXT; subsequent references refer to the value of J, which is assigned the value of EXT at the time of allocation.

c. ALLOCATE RATES SET(S) IN (TABLE);
Allocates space in the storage area corresponding to the area variable TABLE for a generation of the array RATES. The pointer S is set to identify the point within TABLE at which RATES is allocated.

The Assignment Statement

Function:

The assignment statement is used to evaluate an expression and to assign its value to one or more target variables; the target variables may be scalar, array, or structure variables. The target variables may be indicated by pseudo-variables.

General format is shown in Figure 4.

Syntax rule:

In Options 1, 2, and 3 the target variables must be respectively scalars, arrays, and structures. Note that an array of structures is treated as an array.

General rules:

1. Aggregate assignments (Options 2 and 3) are expanded into a series of scalar assignments according to rules 5 through 8.

2. A scalar assignment is performed as follows:

```
Option 1 (Scalar Assignment)

{scalar-variable}     [,scalar-variable]
{pseudo-variable}     [,pseudo-variable]     ... =   scalar-expression;

Option 2 (Array Assignment)

{array-variable }     [,array-variable ]             {structure-expression [,BY NAME]}
{               }     [                ]     ... =   {array-expression [,BY NAME]    };
{pseudo-variable}     [,pseudo-variable]             {scalar-expression              }

Option 3 (Structure Assignment)

{structure-variable}  [,structure-variable]          {structure-expression [,BY NAME]}
{                  }  [                   ]  ... =   {                               };
{pseudo-variable   }  [,pseudo-variable   ]          {scalar-expression              }
```
Figure 4. General Format for the Assignment Statement

a. Subscripts of the target variables, and the second and third arguments of SUBSTR pseudo-variable references, are evaluated from left to right.

b. The expression on the right-hand side is then evaluated.

c. For each target variable (in left to right order), the expression is converted to the characteristics of the target variable according to the rules in "Expressions" in Chapter 3 (except that whenever a conversion of arithmetic base is involved, the value is converted directly to the precision of the target variable). The converted value is then assigned to the target variable.

3. The following rules apply to string scalar assignment:

a. If the target variable is a fixed-length string, the expression value is truncated on the right if it is too long or padded on the right (with blanks for character string, zeros for bit strings) if the value is too short. (Note that a string pseudo-variable is considered to be a fixed-length string). The resulting value is assigned to the target.

b. If the target is a VARYING string and the value of the expression is longer than the maximum length declared for the variable, the value is truncated on the right. The target string obtains a current length equal to its maximum length.

c. If the target is a VARYING string and the value of the expression is not greater than the maximum length, the value is assigned; the target string obtains a current length equal to the length of the value.

4. The following rules apply to assignments other than string:

a. If the target is an area variable, the expression must be an area variable or function. All unfreed allocations in the target area are freed. A sequence of allocations and freeings is then, effectively, performed in the target area for generations corresponding to the significant allocations in the source area; these operations are performed in precisely the same order as the significant allocations were allocated and, where appropriate, freed. The AREA condition is raised by the assignment if any such allocation in the target area raises the AREA condition. Finally, the value of each allocation (which has not been freed) in the source area is assigned to the corresponding allocation in the target area.

b. If the target is a pointer variable, the expression can only be a pointer (or offset) variable or a pointer (or offset) function reference. If the expression is of offset type, its value is converted to pointer by an implicit reference to the POINTER built-in function.

c. If the target is an offset variable, the expression can only be an offset (or pointer) variable or an offset (or pointer) function reference. If the expression is of pointer type, its value is converted to offset by an implicit reference to the OFFSET built-in function.

d. If the target is a label variable, the expression can only be a label variable or label constant. Environmental information is always assigned to the label variable.

e. If the target is an event variable, the expression can only be an event variable. The assignment is uninterruptable, and it involves both the completion and status values; i.e., no other operations will take place (for example, in other tasks) while the assignment is being performed. An event variable does not become active when it has an active event variable assigned to it. It is an error to assign to an active event variable.

f. If the target is a STATUS pseudovariable, a value can be assigned whether or not the event variable is active. It is an error to assign to a COMPLETION pseudovariable if the named event variable is active.

5. The first target variable in an aggregate assignment is known as the master variable. If the master variable is an array, then an array expansion (Rule 6) is performed; otherwise, a structure expansion (Rules 7 and 8) is performed. The generated assignment statements must satisfy the syntax rules. The CHECK condition for assignment to a target variable is not raised during the assignment; it is raised (when suitably enabled) after the assignment is complete. Such CHECK conditions are raised in the written order of the enabled identifiers. In the case of BY NAME assignment, the CHECK condition for the target variable is raised regardless of whether any value is assigned to an item. The label prefix of the original statement is applied to a null statement preceding the other generated statements.

6. In Option 2, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop of the form:

```
LABEL: DO j1 = LBOUND(master-variable,1) TO
               HBOUND(master-variable,1);

       DO j2 = LBOUND(master-variable,2) TO
               HBOUND(master-variable,2);
        .
        .
        .

       DO jn = LBOUND(master-variable,n) TO
               HBOUND(master-variable,n);

       generated assignment statement

       END LABEL;
```

In this expansion, n is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy integer variables j1 to jn. If an array operand appears with no subscripts, it will only have the subscripts j1 to jn; if cross-section notation is used, the asterisks are replaced by j1 to jn. If the original assignment statement (which may have been generated by Rule 7 or Rule 8) has a condition prefix, the generated assignment statement is given this condition prefix. If the original assignment statement (which may have been generated by Rule 8) has a BY NAME option, the generated assignment statement is given a BY NAME option. If the generated assignment statement is a structure assignment, it is expanded as given below.

7. In Option 3, where the BY NAME option is not specified, the following rules apply:

a. None of the operands can be arrays, although they may be structures that contain arrays.

b. All of the structure operands must have the same number, k, of immediately contained items.

c. The assignment statement (which may have been generated by Rule 6) is replaced by k generated assignment statements. The ith generated assignment statement is derived from the original assignment statement by replacing each structure operand by its ith contained item; such generated assignment statements may require further expansion according to Rule 6 or

Rule 7. All generated assignment statements are given the condition prefix of the original statement.

8. In Option 3, where the BY NAME option is given, the structure assignment, which may have been generated by Rule 6, is expanded according to steps a through d below. None of the operands can be arrays.

   a. The first item immediately contained in the master variable is considered.

   b. If each structure operand and target variable has an immediately contained item with the same identifier, an assignment statement is generated as follows: the statement is derived by replacing each structure operand and target variable with its immediately contained item that has this identifier. If any structure contains no such identifier, no statement is generated. If the generated assignments is a structure or array-of-structures assignment, BY NAME is appended. All generated assignment statements are given the condition prefix of the original assignment statement.

   c. Step b is repeated for each of the items immediately contained in the master variable. The assignments are generated in the order of the items contained in the master variable.

   d. Steps a through c may generate further array and structure assignments. These are expanded according to Rules 6 through 8.

Examples:

1. Suppose that the following three structures have been declared:

```
1 ONE                 1 TWO
  2 PART1               2 PART1
    3 RED                 3 RED
    3 WHITE               3 GREEN
    3 BLUE                3 WHITE
  2 PART2               2 PART2
    3 GREEN               3 BLUE
    3 YELLOW              3 YELLOW
    3 ORANGE(3)           3 ORANGE(3)
  2 PART3
    3 BLACK
    3 WHITE
```

```
1 THREE
  3 PART1
    5 BLACK
    5 WHITE
    5 RED
  3 PART2
    5 YELLOW
    5 WHITE
    5 ORANGE(3)
    5 PURPLE
```

Consider the following assignment:

```
ONE = TWO - 2 * THREE, BY NAME;
```

By Rule 8 this generates:

```
ONE.PART1 = TWO.PART1 - 2 *
  THREE.PART1, BY NAME;

ONE.PART2 = TWO.PART2 - 2 *
  THREE.PART2, BY NAME;
```

Applying Rule 8 again, these statements are replaced by:

```
ONE.PART1.RED = TWO.PART1.RED
  - 2 * THREE.PART1.RED;

ONE.PART1.WHITE = TWO.PART1.WHITE
  - 2 * THREE.PART1.WHITE;

ONE.PART2.YELLOW = TWO.PART2.YELLOW
  - 2 * THREE.PART2.YELLOW;

ONE.PART2.ORANGE = TWO.PART2.ORANGE
  - 2 * THREE.PART2.ORANGE;
```

The final assignment is expanded according to Rule 6.

2. The following example illustrates array assignment (Option 2):

```
Given the array A      2   4
                       3   6
                       1   7
                       4   8

and the array B        1   5
                       7   8
                       3   4
                       6   3
```

Consider the assignment statement:

```
    A = (A+B)**2-A(1,1);
```

After execution, A has the value

```
 7  74
93 189
 9 114
93 114
```

Note that the new value for A(1,1), which is 7, is used in evaluating the expression for all other elements.

3. The following example illustrates string assignment:

   Given:

   A is a fixed-length string whose value is 'XZ/BQ'.
   B is a varying-length string of maximum length 8 whose value is 'MAFY'.
   C is a fixed-length string of length 3.
   D is a varying-length string of maximum length 5.

   Then in the statement:

   C=A, the value of C is 'XZ/'.
   C='X', the value of C is 'Xbb'.
   D=B, the value of D is 'MAFY'.
   D=SUBSTR(A,2,3)||SUBSTR(A,2,3), the value of D is 'Z/BZ/'.
   SUBSTR(A,2,4)=B, the value of A is 'XMAFY'.
   SUBSTR(B,2,2)='R', the value of B is 'MRbY'.
   SUBSTR(B,2)='R', the value of B is "MRbb'.

The BEGIN Statement

Function:

The BEGIN statement is the heading statement of a begin block.

General format:

```
BEGIN [OPTIONS(option-list)];
```

Syntax rule:

The syntax of the "option list" is implementation-defined.

General rules:

1. A BEGIN statement is used in conjunction with an END statement.

2. See Chapter 1 for a discussion of blocks.

Examples:

```
1.  ON OVERFLOW BEGIN;
                   .
                   .
                   .
                 END;

2.  (SIZE): Q: PROCEDURE;
              .
              .
              .
          (NOSIZE): A: BEGIN;
                          .
                          .
                          .
                        END;
                   .
                   .
                   .
                 END;
```

The SIZE condition is enabled with the prefix to the PROCEDURE statement. This enabling is negated throughout the begin block with the prefix NOSIZE. On exit from the begin block, SIZE errors are again enabled because statements again are in the scope of the SIZE prefix.

The CALL Statement

Function:

The CALL statement invokes a procedure and causes control to be transferred to a specified entry point of the procedure.

General format:

```
CALL entry-name

[(argument [,argument] . . .)]

     [TASK [(scalar-task-name)]]
     [EVENT (scalar-event-name)]
     [PRIORITY (scalar-expression)];
```

Syntax rules:

1. The entry name, which can be a generic name, represents the entry point of the procedure invoked.

2. An argument is an expresion, entry name, file name, or file parameter

3. The TASK, EVENT, and PRIORITY options can appear in any order.

General rules:

1. The TASK, EVENT, and PRIORITY options, when used alone or in any combination, specify that the invoked and invoking procedures are to be executed asynchronously.

2. When the TASK option is used, the task name, if given, is associated with the task created by the CALL. Reference to this name enables the priority of the task to be controlled at some other point by the use of the PRIORITY pseudo-variable and built-in function.

3. When the EVENT option is used, the event name is associated with the completion of the task created by the CALL statement. Another task can then wait for completion of this created task by specifying the event name in a WAIT statement.

   Upon execution of the CALL statement, the event variable is made active, and the completion value is set to '0'B and the status value to 0. Upon termination of the created task, the completion value is set to '1'B and, unless the task has been terminated by a RETURN or END statement, the status is set to 1 if still zero.

4. If the PRIORITY option is used, the expression in the PRIORITY option is evaluated to an integer m, of an implementation-defined precision (n,0). The priority of the named task is then made m relative to the task in which the CALL is executed.

   If a CALL statement with the EVENT or TASK option does not have the PRIORITY option, the priority of the invoked task is made equal to that of the task variable in the TASK option, if there is one, or else made equal to the priority of the invoking task.

5. Expressions in these options, as well as any argument expressions, are evaluated in the task in which the call is executed. This includes execution of any on-units entered as the result of the evaluations.

6. The environment of the invoked procedure is established after evaluation of the expressions named in Rule 5, and before the procedure is invoked.

Examples:

1.
```
CALL CRITICAL_PATH (A,B*C,D);
                    .
                    .
                    .
CRITICAL_PATH:   PROCEDURE(ALPHA,BETA,
    GAMMA);
                    .
                    .
                    .
                    END;
```

2.
```
CALL PAYROLL (NAME, DATE, HRRATE);
```

3.
```
CALL PRINT (A,B) TASK (T2) EVENT (ET2)
PRIORITY (-2);
```

The CLOSE Statement

Function:

The CLOSE statement dissociates the named file from the data set with which it was associated by opening in the current task.

General format:

```
CLOSE options-group [,options-group]...;
```

Following is the format of "options group":

```
FILE(filename) [IDENT(scalar-argument)]
```

General rules:

1. The options may appear in either order within an options group.
2. The FILE(filename) option specifies which file is to be closed. It must appear once in each options group. Several files can be closed by one CLOSE statement.
3. A closed file can be reopened.
4. Closing an unopened file, or an already closed file, has no effect.
5. The CLOSE statement cannot be used to close a file in a task different from the one that opened the file.
6. If a file is not closed by a CLOSE statement, it is automatically closed at the completion of the task in which it was opened.
7. All I/O event variables associated with operations on the file that have not been completed before the file is closed are set complete, with a status value of 1 if not already non-zero.
8. A CLOSE statement unlocks all records in the file.
9. The argument in the IDENT option is used as follows:

Input Files: The argument must be a character-string variable that may be subscripted. The data set is examined for an identifying user label, which is then assigned to the variable. The label will be a trailer label, unless the file is a BACKWARDS file, in which case it will be a header label. If there is no label, a null string will be assigned.

Output Files: The argument is an expression. Its character-string value is placed with the data set as a trailer label.

Update Files: The argument must be a character-string variable that may be subscripted. The data set is examined for an identifying label, which is then assigned to the variable. The label will be a trailer label.

Examples:

1. CLOSE FILE (MASTER);

   The file, MASTER, is closed, and the facilities allocated to it are released.

2. CLOSE FILE (TABLEA), FILE (TABLEB);

   The two files, TABLEA and TABLEB are closed in the same way as MASTER, in the preceding example.

The DECLARE Statement

See "The DECLARE Statement" in Chapter 4.

The DELAY Statement

Function:

The DELAY statement causes execution of the controlling task to be suspended for a specified period of time.

General format:

```
DELAY (scalar-expression);
```

General rules:

Execution of the DELAY statement causes the scalar expression to be evaluated and converted to an integer n and the task to be suspended for n milliseconds.

Execution resumes after n milliseconds only if the controlling task is of sufficiently high priority to be selected in preference to all other ready tasks.

Example:

```
DELAY (10);
```

The controlling task is suspended for ten milliseconds.

The DELETE Statement

Function:

The DELETE statement deletes a record from an UPDATE file.

General format:

```
DELETE option-list ;
```

Following is the format of "option list":

```
FILE(filename) [KEY(scalar-expression)]
    [EVENT(scalar-event-variable)]
```

General rules:

1. The options may appear in any order.
2. The FILE(filename) option specifies the UPDATE file; it must be specified.
3. The KEY option must be specified if the file is a DIRECT UPDATE file; it cannot be specified otherwise. The expression is converted to a character string and determines which record is to be deleted.
4. If the file is a SEQUENTIAL UPDATE file, the record to be deleted is the last record that was read.
5. The EVENT option allows processing to continue while the record is being deleted. It cannot be specified for SEQUENTIAL BUFFERED files. The completion part of the event-variable is given the value '0'B until the execution of the DELETE is complete, at which time it is given the value '1'B. The execution of a DELETE statement with an EVENT option is considered complete only after a WAIT statement naming the event has been executed.
6. The DELETE statement unlocks a record only if that record had been locked in the same task in which the DELETE appears.
7. The DELETE statement can cause implicit opening of a file.

Example:

```
DELETE FILE(ALPHA) KEY (DKEY);
```

This statement causes the record identified by DKEY to be deleted from the data set associated with the file ALPHA. If the record was previously locked in the same task, it is unlocked.

The DISPLAY Statement

Function:

The DISPLAY statement causes a message to be displayed to the machine operator. A response may be requested.

General format:

Option 1.

```
DISPLAY (scalar-expression);
```

Option 2.

```
DISPLAY (scalar-expression)
REPLY (scalar-character-variable)
[EVENT (scalar-event-variable)];
```

Syntax rule:

REPLY and EVENT may appear in either order.

General rules:

1. Execution of the DISPLAY statement causes the scalar expression to be evaluated and, where necessary, converted to a varying character string of implementation-defined maximum length. This character string is the message to be displayed.

2. In Option 2, the character variable receives a string that is a message to be supplied by the operator.

3. In Option 2, if the EVENT option is not specified, execution of the task is suspended until the operator's message is received. In option 1, execution continues uninterrupted.

4. If the EVENT (event-variable) option is given, execution will not wait for the reply to be completed before continuing with subsequent statements. the completion part of the event variable will be given the value '0'B until the reply is completed, when it will be given the value '1'B. The reply is considered complete only after the execution of a WAIT statement naming the event.

Example:

```
DISPLAY ('END OF JOB');
```

This statement causes the message, "END OF JOB" to be displayed.

The DO Statement

Function:

The DO statement delimits the start of a DO group and may specify repetitive execution of the statements within the group.

General format is shown in Figure 5.

Syntax rules:

1. The "variable" in Option 3 is a subscripted or unsubscripted scalar variable. It cannot be a task or cell variable nor an active event variable, nor can it be an area variable that contains any of those.

```
Option 1.

                              DO;

Option 2.

                DO WHILE (scalar-expression);

Option 3.

                   ⎧pseudo-variable⎫
                DO ⎨               ⎬ = specification [,specification]...;
                   ⎩variable       ⎭

                A specification has the following format:

               ⎡TO expression2      [BY expression3]⎤
expression1    ⎢                                    ⎥[WHILE (expression4)]
               ⎣BY expression3      [TO expression2]⎦
```

Figure 5. General Format for the DO Statement

2. Each "expression" in the specification list is a scalar expression.

3. If the BY clause is omitted from the specification and the TO clause appears, expression3 is assumed to be one (1).

4. If the TO clause is omitted from the specification and the BY clause appears, the iteration is performed until terminated by the WHILE clause, if present, or by some other statement within the group.

5. If both TO expression2 and BY expression3 are omitted, this form of the specification implies a single execution of the DO group with the control variable having the value of expression 1 or it implies no execution if the WHILE statement is false.

6. If the variable in Option 3 is not a string variable or a real arithmetic variable, the TO and BY clauses cannot be used.

General rules:

1. In Option 1, the DO statement delimits the start of a DO group.

2. In Option 2, the DO statement delimits the start of a DO group and specifies repetitive execution defined by the following:

```
LABEL: DO WHILE (expression);
       statement 1
       .
       .
       .
       statement n
       END;
NEXT:  statement
```

The above is exactly equivalent to the following expansion:

```
LABEL: IF (expression) THEN; ELSE GO TO
       NEXT;
       statement 1
       .
       .
       .
       statement n
       GO TO LABEL;
NEXT:  statement
```

3. In Option 3, the DO statement delimits the start of a DO group and specifies controlled repetitive execution defined by the following:

```
LABEL:  DO variable (a1,...,an)=
          expression1
          TO expression2
            BY expression3
          WHILE (expression4);
          statement-1
          .
          .
          .
          statement-m
LABEL1: END;
NEXT:   statement
```

This is exactly equivalent to the following expansion:

```
LABEL:  temp1=a1;
        .
        .
        .
        tempn=an;
        e1=expression1;
        e2=expression2;
        e3=expression3;
        v=e1;
LABEL2: IF (e3>=0)&(v>e2)|
           (e3<0)&(v<e2)
           THEN GO TO NEXT;
        IF (expression4) THEN;
          ELSE GO TO NEXT;
          statement-1
          .
          .
          .
          statement-m
LABEL1: v=v+e3;
        GO TO LABEL2;
NEXT:   statement
```

In the above expansion, $a_1,...,a_n$ are expressions that may appear as subscripts of the control variable; $temp_1...temp_n$ are compiler-created integer variables to which the expression values are assigned; $\underline{v}$ is equivalent to "variable" with the associated "temp" subscripts; "e1," "e2," and "e3" are compiler-created variables having the attributes of "expression1," "expression2," and "expression3," respectively. In the simplest cases, there are no subscripts (i.e., n=0) and the first statement in the expansion is therefore e1=expression1.

Additional rules for the above expansion follow:

a. The above expansion only shows the result of one "specification." If the DO statement contains more than one "specification," the statement labeled NEXT is the first statement in the expansion for the next "specification." The second expansion is analogous to the first expansion in every respect. Thus, if a second "specification" appeared in the DO statement (with expression1 through expression4 represented by expression5 through expression8), the second expansion would look like this:

```
NEXT:    e5=expression5;
         .
         .
         .
         v=e5;
LABEL3:  IF ... THEN GO TO NEXT1;
         IF (expression8) THEN;
           ELSE GO TO NEXT1;
         statement-1
         .
         .
         .
         statement-m
LABEL4:  v=v+e7;
         GO TO LABEL3;
 NEXT1:  statement
```

b. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in the expansion is omitted.

c. If "TO expression2" is omitted, the statement "e2=expression2" and the IF statement identified by LABEL2 are omitted.

d. If both "TO expression2" and "BY expression3" are omitted, all statements involving e2 and e3, as well as the statement GO TO LABEL2, are omitted.

e. Although the above expansions show a specific order, in which the BY and TO clauses are evaluated, no specific ordering is defined by the language.

4. The WHILE clause in Options 2 and 3 specifies that before each associated execution of the DO group, the expression is evaluated and, if necessary, converted to give a bit-string value. If any bit in the resulting string has the value '1', the iteration continues. If all bits have the value '0', the iterations associated with the current specification are terminated.

5. In the specification list, in Option 3, expression1 represents the starting value of the control variable. Expression3 represents the increment to be added to the control variable after each iteration of the statements in the DO group. Expression2 represents the terminating value of the control variable. Iteration terminates as soon as the value of the control variable passes its terminating value. When the last specification is completed, control passes to the statement following the DO group.

6. Control may, under any circumstances, be transferred into a DO group from outside the DO group if the DO group is delimited by the DO statement in Option 1; that is, iteration is not specified. If the DO group is iterative, a GO TO statement can transfer control to a statement inside the group only if the GO TO specifies an abnormal return from a block that has been activated from within the DO group.

7. The effect of allocating or freeing the control variable is undefined.

Examples:

1. DO INDEX = Z WHILE (A>B), 5 TO 10 WHILE (A = B), 100;

2. DO I = 1 TO 9,11 TO 20;

3. DO WHILE (P);

4. DO;

5. DO WHILE (TAX-DEDCT < ESTTAX * 4);

6. DO COMPLEX(X,Y) = 0 BY 1+1I WHILE (X<10);

The END Statement

Function:

The END statement terminates blocks and DO-groups.

General format:

END [label];

General rules:

1. If a label follows END, the END statement terminates the unclosed block or DO-group that is headed by the nearest preceding heading statement having that label; it also terminates all unclosed blocks and DO-groups that are physically within that block or group.

2. If a label does not follow END, the END statement terminates that group or block headed by the nearest preceding DO, BEGIN, or PROCEDURE statement for which there is no other corresponding END statement.

3. If control reaches an END statement terminating a procedure, it is treated as a RETURN statement.

4. If control reaches an END statement which terminates a BEGIN block that is an on-unit, control is returned to the point specified for that particular interrupt.

5. If a label follows END, that label may not be an element of a label array.

For examples, see "Use of the END Statement," in Chapter 1.

The ENTRY Statement

Function:

The ENTRY statement specifies a secondary entry point to a procedure.

General format:

```
{entry-name:} ... ENTRY
    [(parameter [,parameter]...)]
    [data-attributes];
```

General rules:

1. The parameters are names that specify the parameters of the entry point. When the entry is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point.

2. The data attributes with an ENTRY statement are the arithmetic, string, AREA, OFFSET, and POINTER attributes. The data attributes specify the characteristics of the value returned by the procedure when invoked as a function by any of the entry names. The value specified in the RETURN statement of the invoked entry is converted, if necessary, to conform to the specified data attributes.

3. If an ENTRY statement has more than one label, each label is interpreted as if it were a single entry name for a separate ENTRY statement having the same parameter list. If data attributes are specified, they apply to all entry names. If no data attributes are specified, arithmetic defaults are applied separately to each name, depending upon the initial letter of the identifier.

   Consider the statement:

   ```
   A:I: ENTRY;
   ```

   This statement is equivalent to:

   ```
   A:  ENTRY;
   I:  ENTRY;
   ```

   If the entry point were invoked by a function reference to A, a floating-point value will be returned; if to I, a fixed-point value.

4. An ENTRY statement cannot be internal to a begin block, nor can it be internal to a DO group that specifies iteration.

5. A condition prefix cannot be prefixed to an ENTRY statement.

The EXIT Statement

Function:

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task. If the EXIT statement is executed in a major task, it is equivalent to a STOP statement.

General format:

```
EXIT;
```

General rule:

If an EXIT statement is executed in the major task, the FINISH condition is raised in that task. On normal return from the FINISH on-unit, the program is terminated. An EXIT statement executed in any other task terminates the task and its descendants. The event variables associated with these tasks are set complete, and their status values are set to 1 (unless already non-zero).

The FORMAT Statement

Function:

The FORMAT statement specifies a format list for use with data transmitted under edit direction.

General format:

```
label: [label:]...FORMAT(format-list);
```

Syntax rules:

1. The "format list" is as described for use with an edit-directed data specification (see "Format Lists" in Chapter 7).

2. At least one "label" is required. It is the name of a statement label appearing in a remote format item.

General rules:

1. A GET or PUT statement may include a remote format specification, R, in the format list of an edit-directed data specification. That portion of the format list covered by the R format item must be specified in a FORMAT statement with a corresponding statement label.

2. The remote format item and the FORMAT statement must be internal to the same block.

3. A FORMAT statement encountered in sequential flow of control is treated as a no-operation.

4. It is an error to attempt to transfer control to a FORMAT statement by means of a GO TO statement.

5. The CHECK condition is never raised for the label of a FORMAT statement, whether the label is encountered in a format list or in sequential flow of control.

Example:

```
COMMON: FORMAT (A(5), F(5,2), X(3),
   F(10,0));
```

The FREE Statement

Function:

The FREE statement causes the storage allocated for specified based or controlled variables to be freed. For controlled variables, the next most recent allocation in the task is made available, and subsequent references in the task to the identifier refer to that allocation.

General formats:

Option 1

```
FREE controlled-variable
    [,controlled-variable]...;
```

Option 2

```
FREE [locator-qualifier->]
    based-variable
    [IN scalar-area-variable]
    [,[locator-qualifier->]
    based-variable
    [IN scalar-area-variable]]...;
```

Syntax rules:

1. In Option 1, the "controlled variable" must be an unsubscripted, level-one controlled variable.

2. In Option 2, the "based variable" must be an unsubscripted, level-one based variable. See "Locator Qualification," Chapter 2, for more details of locator qualifiers.

3. The forms of Option 1 and Option 2 can be combined in the same FREE statement.

General rules:

1. Controlled storage allocated in a task cannot be freed by a descendant task.

2. If a specified controlled identifier has no allocated storage at the time the FREE statement is executed, no attempt is made to free the storage.

Rules 3 and 4 apply only to Option 2.

3. A based variable can be used to free storage only if that storage has been allocated by a based variable having identical data attributes, including values of bounds, lengths, and area sizes.

4. An IN option must be specified or implied if -- and only if -- the generation to be freed was allocated in an area; the IN option must specify the area in which the generation was allocated. The effect of the FREE statement is to make the relevant storage available for subsequent allocation by an ALLOCATE statement which names the same area in the IN option. If the reference to the variable to be freed is pointer-qualified by the POINTER built-in function (either explicitly, or implicitly by the appearance of an offset as the pointer qualifier), and the IN option is absent, the statement is executed as if it contains the IN option naming the area which is the second argument of the POINTER built-in function. Unless allocation has been in an area, the FREE statement cannot include an IN option nor can an IN option be implied by use of an offset variable.

Examples:

1.
```
DCL A AREA, O OFFSET (A), V BASED (O);
FREE V;
```

   The FREE statement is equivalent to
```
FREE POINTER (O, A)->V IN (A);
```

2. The following excerpt from a procedure illustrates the FREE statement in conjunction with an ALLOCATE statement:

```
   DECLARE A(100) INITIAL ((100)0)
       CONTROLLED , C(100), X(100);
                  .
                  .
                  .
    ALLOCATE A;
                  .
                  .
                  .
    C=A;
                  .
                  .
                  .
    FREE A;
```

3. In the example below, it is assumed the declarations specified in Example 4 of the ALLOCATE statement apply.

   ```
   FREE VALUE;
   ```

   Frees that portion of storage which is occupied by the generation of VALUE identified by pointer P.

   ```
   FREE T -> GROUP;
   ```

   Frees that portion of storage which is occupied by the generation of GROUP identified by pointer T. The value J is used to determine the dimensions of PTS and VALUES.

The GET Statement

Function:

The GET statement normally causes values from a data set to be assigned to variables specified in a data list. Alternatively, the values may come from a character-string variable.

General format:

```
   GET option-list ;
```

Following is the format of "option list":

```
┌                                        ┐
│FILE    (filename) [COPY]                │
│        [SKIP[(scalar-expression)]]      │
│STRING  (scalar-character-string-        │
│        variable)                        │
└                                        ┘
         [data-specification];
```

General rules:

1. If neither the FILE(filename) option nor the STRING(character-string-name) option appears, the file option FILE(SYSIN) is assumed.

2. The data specification must appear unless the SKIP option is specified.

3. The options may appear in any order.

4. The filename refers to a file which has been associated, by opening, with the data set which is to provide the values. It must be a STREAM INPUT file.

5. The "scalar-character-string-variable" refers to the character string that is to provide the data to be assigned to the data list. This name may be a reference to a character string built-in function. Each GET operation using this option always begins at the beginning of the specified string. If the number of characters in this string is less than the total number of characters implied by the data specification, the ERROR condition is raised.

6. When the STRING option is used under data-directed transmission, the ERROR condition is raised if an identifier within the string does not have a match within the data specification.

7. For the rules concerning data specification see "Data Lists", Chapter 7.

8. If the FILE (filename) option refers to a file that is not open in the current task, the file is implicitly opened in the task for stream input transmission.

9. The COPY option, which may only be used with the file option, specifies that the source data, as read, is to be written, without alteration, on the standard installation print file.

10. The SKIP option, which may only be used with the file option, causes a new current line to be defined for the data set. The expression, if present, is converted to an integer _w_, which must be greater than zero. The data set is positioned at the start of the _w_th line relative to the current line. If the expression is omitted, SKIP(1) is assumed. The SKIP option always is executed before any data is transmitted.

Examples:

1. GET LIST (A,B,C);

Specifies the list-directed transmission of the values to be assigned to A, B and C from the file SYSIN.

2. GET FILE (BETA) EDIT (X,Y,Z) (A(5), F(5,2), A(10));

   Specifies the edit-directed transmission of the values assigned to X, Y and Z from file BETA.

<u>The GO TO Statement</u>

Function:

The GO TO statement causes control for a task to be transferred to the specified statement within the task.

General format:

```
{GO TO }{label-constant;         }
{GO TO }{scalar-label-variable;  }
```

General rules:

1. If a label variable is specified, the GO TO statement has the effect of a multi-way switch. The value of the label variable is the label of the statement to which control is transferred.

   Since the label variable may have different values at each execution of the GO TO statement, control may not always pass to the same statement. (Example 2 illustrates a GO TO statement used as a multi-way switch.)

2. A GO TO statement cannot pass control to an inactive block or to another task.

   A GO TO statement cannot transfer control from outside a DO group to a statement inside the DO group if the DO group specifies iteration except in a case in which the GO TO specifies an abnormal return from a block that has been activated from within the DO group.

3. A GO TO statement that transfers control from one block (D) to a dynamically encompassing block (A) has the effect of terminating block D, as well as all other blocks that are dynamically descendant from block A. On-units are reestablished, and automatic variables are freed in the same way as if the blocks were terminated normally. When a GO TO statement transfers control out of a procedure invoked as a function, the evaluation of the expression that contained the corresponding function reference is discontinued, and control is transferred to the specified statement.

4. A GO TO cannot terminate any block activated during a prologue or during execution of an ALLOCATE statement.

Examples:

1.
```
GO TO A234;
    .
    .
    .
A234: ...
```

2. The following example illustrates a GO TO statement that effectively is a multi-way switch:

```
                .
                .
                .
         DECLARE L LABEL  (L1,  L2)  INITIAL
         (L2);
         GO TO MEET;
   L1:   X = Y - 1;
         L = L2;
         GO TO MEET;
   L2:   Y = X -1;
         L = L1;
   MEET: CALL FUDGE (X, Y, Z);
         IF Z = LIMIT THEN GO TO L;
                .
                .
                .
```

3. The following procedure illustrates use of the GO TO statement with a subscripted label variable to effect a multi-way switch:

```
CALC: PROCEDURE (N1, N2);
      DECLARE SWITCH(3) LABEL INITIAL
      (CALC1, CALC2, CALC3);
      I=MOD(N1+N2,3)+1;
      GO TO SWITCH (I);
CALC1: ...
      .
      .
      .
      RETURN;
CALC2: ...
      .
      .
      .
      RETURN;
CALC3: ...
      .
      .
      .
      END CALC;
```

The IF Statement

Function:

The IF statement specifies evaluation of an expression and conversion to bit string, and a consequent flow of control dependent upon the value of the bit string.

General format:

```
IF scalar-expression THEN unit-1 [ELSE
   unit-2]
```

Syntax rules:

1. Each "unit" is a DO-group, a begin block, or any statement, other than DECLARE, END, ENTRY, FORMAT, or PROCEDURE. The unit may have its own labels and condition prefixes.

2. The IF statement is not itself terminated by a semicolon.

General rules:

1. When the ELSE clause -- ELSE, and its following unit -- is not specified, the scalar expression is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string has the value 1, unit-1 is executed, and control passes to the statement following the IF statement. If all bits have the value 0, unit-1 is not executed, and control passes to the next statement. When the ELSE clause is specified, the expression is similarly evaluated. If any bit is 1, unit-1 is executed, and control passes to the statement following the IF statement. If all bits have the value 0, unit-2 is executed, and control passes to the next statement. The units may contain statements that specify transfer of control and so override these normal sequencing rules.

2. IF statements may be nested, that is, either unit-1 or unit-2, or both, may themselves be IF statements. Each ELSE clause is always associated with the innermost unmatched IF in the same block or DO group; consequently, an ELSE or a THEN with a null statement may be required to specify a desired sequence of control.

3. A condition prefix to an IF statement enables (or disables) the condition only during evaluation of the scalar expression of the IF clause; it is not applicable to either of the THEN or ELSE clauses, which may have their own condition prefixes.

Examples:

```
1.  IF A = Z THEN CALL X(0);
             ELSE CALL X(A);

2.  IF X > Y
         THEN IF Z = W
                THEN L: Y = 1;
                ELSE;
         ELSE (SIZE): Y = A;

3.  IF A THEN GO TO M;
    GO TO N;
```

The LOCATE Statement

Function:

The LOCATE Statement, which applies to BUFFERED OUTPUT files, causes allocation of the specified based variable in a buffer; it may also cause transmission of a based variable previously allocated in a buffer.

General format:

```
LOCATE variable option-list ;
```

Following is the format of "option list":

```
FILE(filename)
   [SET(scalar-pointer-variable)]
   [KEYFROM(scalar-expression)]
```

Syntax rules:

1. The options in the option list may appear in any order.

2. The "variable" must be an unsubscripted level 1 based variable.

General rules:

1. The FILE(filename) option specifies the file involved. This option must appear.

2. Execution of a LOCATE statement causes the specified based variable to be allocated in the buffer. Components of the based variable that have been given the INITIAL attribute, or components specified in REFER options, are initialized. A pointer value is assigned to the pointer variable named in the SET option or, if the SET option is omitted, to the pointer variable specified in the declaration of the based variable. The pointer value identifies the record in the buffer. If the pointer variable is an offset variable, the pointer value is implicitly converted. After execution

of the LOCATE statement in a task, values may be assigned to the based variable for subsequent transmission to the file, which will occur immediately before the next LOCATE, WRITE, or CLOSE operation on the file in the task, at which time the record is freed.

3. If the KEYFROM(expression) option appears, the value of the scalar expression is converted to a character string and is used as the key of the record when it is subsequently written.

4. If the FILE(filename) option refers to a file that is not open in the current task, the file is implicitly opened in the task.

Example:

```
LOCATE ALPHA SET (REC_POINT) FILE
   (BETA);
```

The based variable ALPHA is allocated in a buffer and REC_POINT is set to identify ALPHA in the buffer. Values may subsequently be assigned to ALPHA and the record will be written in the data set associated with file BETA when a subsequent LOCATE or WRITE statement is executed for file BETA or if BETA is closed, either explicitly or implicitly.

The Null Statement

Function:

The null statement is a no-operation.

General format:

```
;
```

Example:

```
     .
     .
     .
ON OVERFLOW;
     .
     .
     .
```

The on-unit is a null statement.

The ON Statement

Function:

The ON statement specifies the action to be taken when an interrupt occurs for the named condition. For a discussion of "enable" and "interrupt," see "Interrupt "Operations" in Chapter 6.

General format:

Option 1

```
ON condition [SNAP] on-unit
```

Option 2

```
ON condition [SNAP] SYSTEM;
```

Syntax rules:

1. The "condition" may be any one of those described in Appendix 3.

2. The "on-unit" is an action specification, and it is either an unlabeled single simple statement (other than BEGIN, DO, END, RETURN, ENTRY, FORMAT, PROCEDURE, or DECLARE) or an unlabeled begin block. It may have a condition prefix. Since the on-unit itself requires a semi-colon, no semi-colon appears in Option 1.

3. The on-unit may not be a RETURN statement, nor may a RETURN statement be internal to the begin block.

General rules:

1. An ON statement, such as in Option 1, must be executed before its effect can be established.

2. The standard action to be taken for all ON-conditions is defined by the language. When an interrupt takes place before an ON statement for that condition has been executed, standard system action is taken. This standard system action is described in Appendix 3. The ON statement in Option 2 specifies that standard system action is to be taken when an interrupt results from the occurrence of the specified condition.

3. The ON statement in Option 1 is a means for the programmer to specify action (other than standard system action), that is, execution of the on-unit, to take place when an interrupt occurs for the specified condition. The on-unit is treated as a procedure internal to the block in which it appears.

4. If SNAP is specified, then when the given condition occurs, a calling trace is listed.

5. Control can reach an on-unit only when an interrupt occurs for the condition associated with this on-unit in an ON statement.

6. If an action specification is established by execution of an ON statement, it remains in effect until it is overriden by another ON statement or REVERT statement specifying the same condition, or until termination of the block in which the ON statement is executed.

7. A single statement on-unit cannot contain a remote format item.

<u>The OPEN Statement</u>

Function:

The OPEN statement associates a filename with a data set and completes the specification of attributes for the file.

General format:

```
OPEN options-group [,options-group]...;
```

Following is the format of "options group":

```
    FILE(filename)
    [IDENT(scalar-argument)]
    [TITLE(scalar-expression)]
    [INPUT | OUTPUT | UPDATE]
    [STREAM | RECORD]
    [DIRECT | SEQUENTIAL]
    [BUFFERED | UNBUFFERED]
    [EXCLUSIVE]
    [KEYED]
    [BACKWARDS]
    [PRINT]
    [LINESIZE (scalar-expression)]
    [PAGESIZE (scalar-expression)]
```

General rules:

1. The INPUT, OUTPUT, UPDATE, STREAM, RECORD, DIRECT, SEQUENTIAL, BUFFERED, UNBUFFERED, EXCLUSIVE, KEYED, BACKWARDS, and PRINT options specify attributes which may augment the attributes specified in the file declaration; the options may repeat attributes specified in a DECLARE statement, but they must not conflict with any declared attributes.

2. The options may appear in any order within a group.

3. The FILE(filename) option specifies which file is to be opened. The option must appear once in each options group. Several files can be opened by one OPEN statement.

4. If a file has been opened in a particular task and not subsequently closed, then re-opening this file in the same task or a descendant task has no effect on the file or its associated data set. All options (including TITLE) are evaluated whether or not they conflict with the options of the previous OPEN, but they are not used. If a file has been opened and subsequently closed, it may be re-opened in the task that originally opened it, but any attempt to open it (or use it) in a descendant of that task -- if the descendant has inherited the file as an open file -- will give undefined results.

5. The "argument" in the IDENT option is used as follows:

   <u>Input files:</u> The argument must be a character-string variable that may be subscripted. The data set is examined for an identifying user label which is then assigned to the variable given as the argument. The label will be a header label unless the file is a BACKWARDS file, in which case it will be a trailer label. If there is no label, a null string will be assigned to the character string variable.

   <u>Output files:</u> The argument is an expression. Its character-string value of the argument is placed with the data set as a header label.

   <u>Update files:</u> The argument must be a character-string variable that may be subscripted. The data set is examined for an identifying label which is then assigned to the variable given as the argument. The label is a header label.

6. If the TITLE(expression) option appears, the expression is converted to a character string which is used in the association of a data set with the file. If the option does not appear, a character string identical to the filename is taken as the identification. In the case of a parameter, the identifier of the original argument passed to the parameter, rather than the identifier of the parameter itself, is used. A data set may be

accessed by two or more files only if all the files are direct.

7. The LINESIZE option can be specified only for a STREAM OUTPUT file. The expression is evaluated, converted to an integer, and used as the length of a line during subsequent operations on the file. New lines may be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is automatically started, and the file is positioned to the start of this new line. If no LINESIZE is given for a STREAM OUTPUT file, an implementation-defined default is supplied.

   The LINESIZE option cannot be specified for an INPUT file. The linesize taken into consideration whenever a SKIP option appears in a GET statement is the linesize that was used to create the data set.

8. The PAGESIZE option can be specified only for a STREAM PRINT file. The expression is converted to an integer and used as the number of lines on a page. During subsequent output to the file, new pages may be started by use of the PAGE format item or PUT statement option. If a page becomes overfilled before action to start a new page is given, the ENDPAGE condition is raised. Default is implementation defined.

Examples:

1. OPEN FILE (ALPHA), FILE (BETA) TITLE ('WORKFILE');

   The files ALPHA and BETA are opened. The data set associated with BETA is identified through use of the name WORKFILE, whereas ALPHA is identified with a data set through use of the name ALPHA.

2. OPEN FILE (MASTER) UPDATE;

   The file MASTER is opened as an UPDATE file. MASTER is the name used to associate a data set with the file.

The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

1. Identifies a portion of program text as a procedure.

2. Defines the primary entry point to a procedure.

3. Specifies the parameters for the primary entry point.

4. Defines any special attributes of the procedure.

5. Specifies the attributes of the value that is returned if the procedure is invoked as a function at the primary entry point.

General format:

```
{entry-name:} ... PROCEDURE
    [ (parameter [, parameter]...)]
    [OPTIONS(option-list)]
    [RECURSIVE] [data-attributes];
```

Syntax rules:

1. The data attributes and the OPTIONS and RECURSIVE options may appear in any order.

2. The syntax of the OPTIONS list is implementation-defined.

General rules:

1. The "parameters" are names that specify the parameters of the entry point. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameters of the invoked entry point (see "Correspondence of Arguments and Parameters" in Chapter 6).

2. OPTIONS specifies a list of options, which depends upon implementation. OPTIONS may be specified for any procedure.

3. RECURSIVE specifies that the procedure may be invoked recursively. This option applies only to the procedure for which it is declared, but not any procedures contained in it. It is an error to invoke a procedure recursively by any of its entry points if it is not given the RECURSIVE option in its PROCEDURE statement.

4. The data attributes permitted with a PROCEDURE statement are the arithmetic, string, AREA, OFFSET, and POINTER attributes. The data attributes specify the characteristics of the value returned by the procedure when invoked as a function at the primary entry point. (This rule applies to each entry name by which the procedure may

be invoked, i.e., each entry name prefixed to the PROCEDURE statement.) The value specified in the RETURN statement of the invoked procedure is converted to the specified data attributes.

5. If a PROCEDURE statement has more than one entry name, the first name is interpreted as the only label on the statement; each subsequent entry name is interpreted as a separate ENTRY statement having an identical parameter list and the same data attributes as written in the PROCEDURE statement. This equivalence is true only after multiple closure has been resolved. Defaults for the data attributes are applied separately for each such entry statement and for the resulting procedure statement. If no data attributes are specified, arithmetic defaults are applied separately to each name, depending upon the initial letter of the identifier.

   For example, the statement:

   ```
   A:I:    PROCEDURE;
   ```

   is effectively the same as:

   ```
   A·  PROCEDURE;
   I:  ENTRY;
   ```

   Since no data attributes are specified in the example, defaults will differ for the two entry names. The equivalance applies only after multiple closure has been resolved.

Example:

```
B:  PROCEDURE;
    DECLARE A RETURNS(FIXED);
          .
          .
          .
    D=A(X,Y);
    END B;
A:  PROCEDURE (B,C) FIXED;
          .
          .
          .
    RETURN (B*C + SIN (P));
    END A;
```

If procedure A is invoked as a function, as it is in procedure B, then when control is returned to B, the expression (B*C + SIN (P)) is evaluated, converted to fixed point, and the value assigned to D in procedure B.

The PUT Statement

Function:

The PUT statement causes the transmission of data and/or the execution of control options. Data items transmitted are the character-string representations of values of expressions that are assigned to a data set or to a designated character-string variable.

General format:

```
PUT option-list ;
```

Following is the format of "option list":

```
[FILE(filename)|STRING
   (scalar-character-string-variable)]

[data-specification][PAGE]
   [SKIP [(expression)]]
   [LINE (expression)]
```

Syntax rule:

The PAGE, SKIP, and LINE options cannot be used with the STPING option.

General rules:

1. If neither the FILE (filename) option nor the STRING (character string name) appears, the file option FILE(SYSPRINT) is assumed.

2. The "filename" refers to a file that has been associated, by opening, with the data set that is to receive the values. It must be a STREAM OUTPUT file.

3. The "scalar-character-string-variable" refers to the character string variable or pseudo-variable that is to receive the values.

   After appropriate conversion, the data specified by the data list is assigned to the string starting at the leftmost character (leftmost specified character in the case of a SUBSTR pseudo-variable). Note that any subsequent PUT statement will cause assignment to begin at the same place. If the string is not long enough to accommodate the data, the ERROR condition is raised.

4. The options may appear in any order. The PAGE and LINE options can be specified for PRINT files only. All of the options take effect before transmission of any values defined by the data specification, if given. Of

the three, only PAGE and LINE may appear in the same PUT statement, in which case, the PAGE option is applied first.

5. The PAGE option causes a new current page to be defined within the data set. If a data specification is present, the transmission of values occurs after the definition of the new page. The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until an ENDPAGE interrupt results in the definition of a new page. A new current page implies line one.

6. The SKIP option causes a new current line to be defined for the data set. The expression, if present, is converted to an integer _w_, which for non-PRINT files must be greater than zero. The data set is positioned at the start of the _w_th line relative to the current line. If the expression is omitted, SKIP(1) is assumed.

   For PRINT files _w_ may be less than or equal to zero; in this case, the effect is that of a carriage return with the same current line. If less than _w_ lines remain on the current page when a SKIP(w) is issued, ENDPAGE is raised.

7. The LINE option causes a current line to be defined for the data set. The expression is converted to an integer _w_. If _w_ specifies the current line of the most recent PUT statement, no new current line is established. If _w_ is greater, blank lines are inserted so that the next line will be the _w_th line of the current page. If more than _w_ lines have already been written on the current page or if _w_ exceeds the limits set by the PAGESIZE option of the OPEN statement or by default , the ENDPAGE condition is raised. If _w_ is less than or equal to zero, it is assumed to be 1.

8. If the FILE(filename) option refers to a file that is not open in the current task, the file is opened implicitly in this task for stream output.

Examples:

1. PUT DATA (A,B,C);

   Specifies the data-directed transmission of the values A, B and C to the file SYSPRINT.

2. PUT FILE (LIST) EDIT (X,Y,Z) (A(10)) PAGE;

   Specifies that a new page is to be defined for the print file LIST. The values of X, Y and Z are placed starting in the first printing position of the new page. Each of the values will use the A(10) format item.

<u>The READ Statement</u>

Function:

The READ statement causes a record to be transmitted from a RECORD INPUT or RECORD UPDATE file to a variable or buffer.

General format:

```
READ option-list ;
```

Following is the format of "option list":

```
FILE (filename)
 ┌INTO (variable)                  ┐
 │SET(scalar-pointer-variable)     │
 └IGNORE(scalar-expression)        ┘

 ┌KEY (scalar-expression)                   ┐
 │KEYTO                                     │
 └ (character-string-scalar-variable)       ┘
 [EVENT (event-scalar-variable)]
  [NOLOCK]
```

General rules:

1. The options may appear in any order.

2. The FILE(filename) option specifies the file from which the record is to be read. This option must appear. If the file specified is not open in the current task, it is implicitly opened in the task.

3. The INTO(variable) option specifies an unsubscripted level 1 variable into which the record is to be read. It cannot be a parameter, nor can it have the DEFINED attribute.

4. The KEY and KEYTO options can be specified for KEYED files only.

5. The KEY(expression) option must appear if the file is DIRECT. The expression is converted to a character string that determines which record is read.

6. The KEYTO(character-string-variable) option may be given only if the file is SEQUENTIAL. It specifies that the key of the record is to be copied into the string variable, which may be a pseudo-variable. This copying follows the rules for character string assign-

ment, and if proper assignment cannot be made, the KEY condition is raised. The key will match that which was specified in the KEYFROM option when the record was written. KEYTO and KEY may not appear in the same READ statement.

7. The EVENT(event-variable) option allows processing to continue while the record is being read or ignored. It may not be specified for SEQUENTIAL BUFFERED files. If the EVENT (event variable) option is given, the event variable will be set active and will be given the value '0'B until the execution is complete, when it will be given the value '1'B. The execution of a READ statement with an EVENT option is considered complete only after the execution of a WAIT statement naming that event variable.

8. Any READ statement referring to an EXCLUSIVE file will cause the record to be locked for access by a given opening of a file unless the NOLOCK option is specified. A locked record cannot be read, deleted, or rewritten by any other task until it is unlocked. Any attempt to read, delete, rewrite, or unlock a record locked by another task results in a wait. Subsequent unlocking can be accomplished by the locking task through the execution of an UNLOCK, REWRITE, or DELETE statement that specifies the same key, by a CLOSE statement, or by completion of task in which the record was locked.

   Note that a record is considered locked only for tasks other than the task that actually locks it; in other words, a locked record can always be read by the task that locked it and still remain locked as far as other tasks are concerned (unless, of course, the record has been explicitly unlocked by one of the above methods).

9. The SET option specifies that the record is to be read into a buffer and that a pointer value is to be assigned to the named pointer variable. The pointer value identifies the record in the buffer.

10. The IGNORE option may be specified for SEQUENTIAL INPUT and SEQUENTIAL UPDATE files. The expression in the IGNORE option is evaluated and converted to an integer. If the value, n, is greater than zero, n records are ignored; a subsequent READ statement for the file will access the (n+1)th record. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an IGNORE(1).

11. A keyed file being accessed sequentially may be positioned by issuing a READ statement with the KEY option. The specified key will be used to identify the record required. Thereafter, records may be read sequentially from that point by use of READ statements without the KEY option. This applies to INPUT and UPDATE files.

    For BUFFERED SEQUENTIAL files, two positioning statements can be used, with the following formats:

```
READ  FILE (filename)
      INTO (variable)
      KEY (expression);

READ  FILE (filename)
      SET (pointer-variable)
      KEY (expression);
```

    For UNBUFFERED SEQUENTIAL files, only the first form shown immediately above can be used, and it may be specified with the EVENT option.

Examples:

1. READ FILE (ALPHA) SET (REC_IDENT);

   The next record from the data set associated with ALPHA is made available and the pointer variable REC_IDENT is set to identify the record in the buffer.

2. READ FILE (BETA) KEY (VALUE) INTO (WORK);

   The record identified by the key VALUE is transmitted from the data set associated with BETA into the variable WORK.

The RETURN Statement

Function:

The RETURN statement terminates execution of the procedure that contains the RETURN statement. If the procedure has not been invoked as a task, the RETURN statement returns control to the invoking procedure. The RETURN statement may also return a value.

General format:

Option 1.

```
    RETURN;
```

Option 2.

```
    RETURN (scalar-expression);
```

General rules:

1. Only the RETURN statement in Option 1 can be used to terminate procedures not invoked as function procedures; control is returned to the point logically following the invocation.

   Option 1 represents the only form of the RETURN statement that can be used to terminate a procedure initiated as a task. If the RETURN statement terminates the major task, the FINISH condition is raised prior to the execution of any termination processes. If the RETURN statement terminates any other task, the completion value of the associated event variable (if any) is set to '1'B, and the status value is left unchanged.

2. The RETURN statement in Option 2 is used to terminate a procedure invoked as a function procedure only. Control is returned to the point of invocation, and the value returned to the function reference is the value of the expression specified converted to conform to the attributes declared for the invoked entry point. These attributes may be explicitly specified at the entry point; they are otherwise implied by the initial letter of the entry name through which the procedure is invoked.

3. If control reaches an END statement corresponding to the end of a procedure, this END statement is treated as a RETURN statement (of the Option 1 form) for the procedure.

Example:

```
     A:  PROCEDURE (X,Y) FIXED;
         DECLARE (X,Y) FLOAT;
         .
         .
         .
         RETURN (X**2+Y**2);
         END;
     B:  PROCEDURE;
         DECLARE A ENTRY RETURNS (FIXED);
         .
         .
         .
         R = A(P,Q);
         .
         .
         .
         END;
```

In the assignment statement (R=A(P,Q);), procedure B invokes procedure A as a function. Procedure B specifies that the scalar expression in the RETURN statement is to be evaluated; since X and Y are floating-point variables and the PROCEDURE statement specifies that the value returned is to be fixed point, the value of the expression is converted to fixed point, and this value is returned to B.

The REVERT Statement

Function:

A REVERT statement specifying a given ON-condition is used to reestablish the action specification for the named condition as it was in the immediate, dynamically encompassing block. In the case of an initial external procedure, standard system action is established.

General format:

```
    REVERT condition;
```

Syntax rule:

The "condition" is any ON-condition (see Appendix 3).

Examples:

```
 A:   PROCEDURE;
      .
      .
      .
ON1:  ON ZERODIVIDE GO TO ERRSPEC;
      .
      .
      .
      CALL B;
      .
      .
      .
      B:  PROCEDURE;
          .
          .
          .
      ON2:ON ZERODIVIDE;
          .
          .
          .
          REVERT ZERODIVIDE;
          .
          .
          .
          END B;
      .
      .
      .
      .
      .
      .
      END A;
```

Unless it is stated otherwise, the condition ZERODIVIDE always is enabled. If division by zero occurs prior to execution of statement ON1, an interrupt with standard system action takes place.

If division by zero occurs after execution of ON1 and prior to execution of statement ON2, an interrupt takes place and control transfers to the statement GO TO ERRSPEC.

If division by zero occurs after execution of ON2 and prior to the REVERT statement, an interrupt takes place effectively with no action.

When the REVERT statement is executed, the effect of the statement ON2 is nullified, and statement ON1 again becomes effective.

The REWRITE Statement

Function:

The REWRITE statement causes replacement of an existing record in a data set referred to by an UPDATE file.

General format:

```
REWRITE option-list ;
```

Following is the format of "option list":

```
FILE(filename) [KEY(scalar-expression)]
    [FROM(variable)]
    [EVENT (event-scalar-variable)]
```

General rules:

1. The options may appear in any order.

2. The FILE(filename) option specifies the file involved. If it refers to a file that is not open in the current task, the file is opened implicitly in this task.

3. The KEY(expression) option must appear if the file is a DIRECT UPDATE file and it cannot appear otherwise. The expression is converted to a character string and determines which record is written.

4. The FROM(variable) option may be given to specify an unsubscripted level 1 variable which is to be used as the source for the record. The FROM(variable) option must be specified for a DIRECT UPDATE or SEQUENTIAL UNBUFFERED UPDATE file. The FROM option can be omitted for SEQUENTIAL BUFFERED UPDATE files only, in which case, the file is updated from the buffer associated with the file.

5. The EVENT (event-variable) option allows processing to continue while the record is being written. It may not be specified for SEQUENTIAL BUFFERED files. If the EVENT (event variable) option is given, the event variable will be made active and will be given the value '0'B until the execution is complete, when it will be given the value '1'B. The execution of a REWRITE statement with an EVENT option is considered complete only after the execution of a WAIT statement naming that event.

6. If the record rewritten is one that was locked in the same task, it becomes unlocked.

Example:

```
REWRITE FILE (ALPHA);
```

The last record read from the data set associated with file ALPHA is rewritten from the buffer.

The SIGNAL Statement

Function:

The SIGNAL statement simulates the occurrence of the named condition and causes an interrupt if the condition is enabled. It may be used to test the action specification of the current ON statement.

General format:

```
SIGNAL condition;
```

Examples:

```
1.    X: PROCEDURE;
         .
         .
         .
    ON1: ON ENDFILE (DATIN) Y,Z = 0;
         .
         .
         .
     S1: SIGNAL ENDFILE (DATIN);
         .
         .
         .
    ON2: ON ENDFILE (DATIN) SYSTEM;
         .
         .
         .
    S2: SIGNAL ENDFILE (DATIN);
        .
        .
        .
        END X;
```

The S1 statement causes an interrupt in the same way as if an attempt to read past a file delimiter had actually occurred. Control is transferred to the statement Y,Z = 0 in the ON1 statement.

When the S2 statement causes an interrupt, control is transferred to the ON2 statement, and standard system action is taken.

2.
```
ON CONDITION (TAX) TAXCT = TAXCT+1;
.
.
.
SIGNAL CONDITION (TAX);
```

The ON statement establishes an action for the programmer-specified condition TAX. This condition can occur only when a SIGNAL statement causes the condition to occur.

The STOP Statement

Function:

The STOP statement causes immediate termination of the major task and all subtasks.

General format:

```
STOP;
```

General rule:

Prior to any termination activity the FINISH condition is raised in the task in which the STOP is executed. On normal return from the FINISH on-unit, all tasks in the program are terminated.

The UNLOCK Statement

Function:

The UNLOCK statement makes the specified locked record available to other tasks for operations on the record.

General format:

```
UNLOCK option-list;
```

Following is the format of "option list":

```
FILE(filename) KEY(scalar-expression)
```

General rules:

1. The options may appear in either order.
2. The FILE(filename) option specifies the file involved, which must have the attributes UPDATE, DIRECT, and EXCLUSIVE. If the file is not open in the current task, it is opened implicitly.
3. In the KEY(expression) option, the "expression" is converted to a character string that determines which record is unlocked.
4. A record can be unlocked only by the task which locked it.

The WAIT statement

Function:

The execution of a WAIT statement within an activation of a block retains control for that activation of that block within the WAIT statement until certain specified events have completed.

General format:

```
WAIT (event [,event]...)
  [(scalar-expression)];
```

Syntax rule:

Each event is an event variable or structure consisting only of event variables.

General rules:

1. Control for a given block activation remains within this statement until, at possibly separate times during the execution of the statement, the condition

   ```
   COMPLETION(event)  = '1'B
   ```

   has been satisfied, for some or all of the events in the list.
2. If the optional expression does not appear, all the event names in the list must satisfy the above condition before control returns to the next statement in this task following the WAIT.

3. If the optional expression appears, the expression is evaluated when the WAIT statement is executed and converted to an integer. This integer specifies the number of events in the list that must satisfy the above condition before control for the block passes to the statement following the WAIT. Of course, if an on-unit entered due to the WAIT is terminated abnormally, control might not pass to the statement following the WAIT.

   If the value of the expression is zero or negative, the WAIT statement is treated as a null statement. If the value of the expression is greater than the number, n, of events in the list, the value is taken to be n. If the statement refers to an aggregate of event variables, then each element in the aggregate contributes to the count.

4. If the event variable named in the list has been associated with a task in its attaching CALL statement, then the condition in Rule 1 will be satisfied on termination of that task.

5. If the event variable named in the list is associated with an I/O operation initiated in the same task as the WAIT, the condition in Rule 1 will be satisfied when the I/O operation is completed. The execution of the WAIT is a necessary part of the completion of an I/O operation. If prior to, or during, the WAIT all transmission associated with the I/O operation is terminated, then the WAIT performs the following action: If the transmission has finished without requiring any I/O conditions to be raised, the event variable is set complete. If the transmission has been terminated but has required conditions to be raised, the event variable is set abnormal, and all the required ON conditions are raised. On return from the last on-unit, the event variable is set complete.

6. The order in which ON conditions for different I/O events are raised is not dependent on the order of appearance of the events in the list. If an ON condition for one event is raised, then all other conditions for that event are raised before the WAIT is terminated or before any other I/O conditions are raised unless an abnormal return is made from one of the on-units thus entered. The order in which I/O ON conditions are raised for an event is implementation-defined. The raising of ON conditions for one event implies nothing about the completion or termination of transmission of other events in the list.

7. If an abnormal return is made from any on-unit entered from a WAIT, the associated event variable is set complete, the execution of the WAIT is terminated, and control passes to the point specified by the abnormal return.

8. If some of the events in the WAIT list are associated with I/O operations and have not been set complete before the WAIT is terminated (either because enough events have been completed or due to an abnormal return), these incomplete events will not be set complete until the execution of another WAIT referring to these events.

Example:

```
PI:  PROCEDURE;
       .
       .
       .
     CALL P2 EVENT(EP2);
       .
       .
       .
     WAIT(EP2);
       .
       .
       .
     END;
```

The CALL statement, when executed, attaches a task whose completion status is associated with the event name EP2. When the WAIT statement is encountered, the execution of the attached task is suspended until the value of COMPLETION(EP2) is '1'B, i.e., until the attached task is completed.

The WRITE Statement

Function:

The WRITE statement transfers a record from a variable in internal storage to a RECORD OUTPUT or DIRECT RECORD UPDATE file.

General format:

```
WRITE option-list ;
```

Following is the format of "option list":

```
FILE(filename) FROM(variable)
   [KEYFROM(scalar-expression)]
   [EVENT(event-scalar-variable)]
```

General rules:

1. The options may appear in any order.

2. The FILE(filename) option, which must appear, specifies the file in which the record is to be written. If the file is not open in the current task, it is opened implicitly in this task.

3. The FROM(variable) option specifies an unsubscripted level 1 variable which is to be written.

4. The expression in the KEYFROM option is converted to character string and associated with the record as its key.

5. The EVENT(event variable) option allows processing to continue while the record is being written. It may not be specified for SEQUENTIAL BUFFERED files. If the EVENT (event variable) option is given, the event variable will be made active and given the value '0'B until the execution is complete, when it will be given the value '1'B. The execution of a WRITE statement with an EVENT option cannot be considered complete until a WAIT statement naming that event has been executed.

Example:

```
WRITE FILE(BETA) FROM(UPDATE)
   KEYFROM(ONKEY);
```

Specifies that the record UPDATE is written as the next record in the data set associated with file BETA. The key identifying the record in the data set is taken from UKEY.

PL/I allows a programmer to alter the text of a source program at compile time. This can be done in the following ways:

1. Modification of a source program for the purpose of changing variable names or for notational convenience.

2. Conditional compilation of sections of the source program. In other words, the user can dictate which sections of his program are to be compiled.

3. Incorporation of strings of text into the source program, where the strings of text reside in a user or system library.

These operations are performed by the preprocessor stage of the compiler.

THE PREPROCESSOR

PREPROCESSOR INPUT AND OUTPUT

The preprocessor interprets compile-time statements and acts upon the source program accordingly. Input to the preprocessor consists of a character string, called the source text, which consists of identifiers and constants; between any two of these, there must be at least one blank, delimiter, comment, or compile-time statement. Compile-time statements are identified by a leading percent sign (%) and are executed upon being encountered by the processor. One or more blanks and/or comments may separate the percent sign from the statement. Note that a percent sign appearing in a character string is considered only to be a character in that string.

Compile-time activity may also be specified by statements in a compile-time procedure. In this case, only the PROCEDURE and END statements require, or can have, percent signs. A compile-time procedure is invoked by a compile-time function reference.

Output from the preprocessor consists of a newly created character string, called the program text, which contains the modified source program text, and which serves as input to the compiler. This new text has been modified by the preprocessor according to the compile-time statements encountered in the source text.

THE PREPROCESSOR SCAN

The preprocessor begins to scan the characters of the source text in a sequential manner. If the source text does not contain a compile-time statement, the preprocessor places the scanned characters into the program text in the same order and form in which they were encountered.

When a compile-time statement is encountered during the scan, it is executed. This execution may cause the sequential scanning and placing of characters to be modified in either of the following ways:

1. The executed compile-time statement may cause the preprocessor to continue the scan from a different point in the source text.

2. The executed compile-time statement may specify to the preprocessor that upon the subsequent encounter of a specified identifier within the source program, that identifier itself is not to be inserted into the program text being generated; rather, the currently assigned value of the identifier (that is, the value assigned by a compile-time statement executed prior to this encounter) is to be placed into the program text (unless this value or part of it, in turn, can be replaced -- see "Rescanning and Replacement" below). Note that compile-time statements themselves are never inserted in the program text; rather, a blank is inserted in place of such compile-time statements.

The preprocessor scan is terminated when an attempt is made to scan beyond the last character in the source text. The resulting program text is a string representing the PL/I program to be compiled.

Rescanning and Replacement

Replacement of a variable or invocation of a compile-time procedure (and subsequent replacement of the function reference) cannot take place until the variable or the entry name has been activated, either by a reference in a %ACTIVATE or %DECLARE statement.

When an activated variable or an activated procedure name is encountered in the source text, its value becomes a candidate for replacement. This value cannot contain percent signs, unmatched quotation marks, or unmatched comment delimiters. The value is then rescanned from left to right to determine whether or not it, or any part of it, can be replaced, at the second replacement level, by another value. If it cannot be replaced, it is inserted into the program text; if it is replaced, the new value, in turn, is rescanned, etc. Thus, insertion of a value into program text takes place only after all possible replacements have been made (see Example 2 below).

Examples:

1. If the source text contained the following statements:

   ```
   % DECLARE A CHARACTER, B FIXED;
   % A = 'B + C';
   % B = 2;
   X = A;
   ```

   then the following would be generated in the program text:

   ```
      X =   2  + C ;
   ```

   In the above example, the first statement is a compile-time DECLARE statement that establishes A and B as compile-time variables with the indicated attributes, and also serves to activate these variables. The second statement is a compile-time assignment statement that assigns the character string 'B + C' to A. The third statement is also a compile-time assignment statement, and assigns the value 2 to B. The fourth statement is a source program statement which assigns A to X. However, since A has been activated for replacement and has been assigned a value, namely, the string 'B + C', the value of A is rescanned for possible further replacement action. This rescanning causes B to be replaced by the value 2. However, since 2 is not a compile-time variable, it cannot be replaced, and the chain of replacements comes to an end. Thus, the source program statement X = A; becomes the program text statement X = 2 + C. Note that a blank is appended to each end of the replacement value when it is written into the program text. Also note that in the examples shown in this chapter all leading blanks of fixed-point values are not shown.

2. The following example illustrates an error because a procedure name and its delimited argument list are not provided at the same replacement level:

   ```
   % DECLARE (A,B,C) CHARACTER,
             D ENTRY (CHARACTER)
             RETURNS (CHARACTER);
   % D: PROCEDURE (E) CHARACTER;
             .
             .
             .
   % END;
   % A = 'D';
   % B = 'X)';
   % C = 'A(B';
     Y = C;
   ```

   In the first scan, the compile-time statements cause the following replacement:

   ```
     Y = A(B;
   ```

   The second scan causes replacement of A, as follows:

   ```
     Y = D(B;
   ```

   In the third scan, since it is done from left to right, the character 'D' is encountered before the character 'B'. Consequently, an attempt would be made to invoke the procedure before the argument list is complete, which would be in error. The complete, delimited argument list (X), (as intended in this coding) would have to be supplied at the same (or an earlier) level of replacement as the entry name D.

Example: Compile-Time Loop Expansion

A programmer may wish, at object-time, to execute the following loop:

```
DO I = 1 TO 10;
Z(I) = X(I) + Y(I);
END;
```

The following program would accomplish the same thing, but without the execution-time requirements of incrementing and testing:

```
% DECLARE I FIXED;
% I = 1;
% LAB:;
Z(I) = X(I) + Y(I);
% I = I + 1;
% IF I<= 10 % THEN % GO TO LAB;
% DEACTIVATE I;
```

The precise effect of each of these statements is detailed below.

The statement % I=1 assigns the value 1 to the compile-time variable I and specifies that, unless the programmer indicates otherwise (note the later appearance of the % DEACTIVATE statement), subsequent occurrences of the identifier I in the source program will result in its replacement in the program text by the string '1'. The % LAB: statement is a compile-time null statement that is used as the transfer target for the % GO TO statement that appears later.

The string 'Z(I) = X(I) + Y(I);' is a source program statement. Initially, the variable I was given the value 1; therefore, the first time that this string is scanned, the string 'Z(1) =X(1) + Y(1);' will be inserted into the program text by the preprocessor. I is then incremented by 1 (% I = I+1;), after which the compile-time IF statement instructs the preprocessor to test the value of I. If I is not greater than 10, the scan is to resume at the compile-time statement labeled LAB; otherwise, the scan is to continue with the text immediately following the % GO TO statement.

The % DEACTIVATE statement is interpreted as follows: subsequent occurrences of the identifier I in the source program are not to be replaced by the string '11' in the program text being formed (note that I has the value 11 at the time the % DEACTIVATE statement is encountered); instead each I will be left unmodified.

As a result of the above compile-time activity, the following PL/I statements are generated into the program text:

```
Z( 1 ) = X( 1 ) + Y( 1 );
Z( 2 ) = X( 2 ) + Y( 2 );
       .
       .
       .
Z( 10 ) = X( 10 ) + Y( 10 );
```

The foregoing statements are the statements that will actually be compiled into executable object code.

COMPILE-TIME VARIABLES

A compile-time variable is an identifier that has been specified in a %DECLARE statement with either the FIXED or CHARACTER attribute. No other attributes can be declared for a compile-time variable. Defaults are applied, however. A compile-time variable declared with the FIXED attribute is also given the attributes DECIMAL and an implementation-defined precision; a CHARACTER compile-time variable is given the VARYING attribute with no maximum length. No contextual or implicit declaration of identifiers is allowed in compile-time statements.

The scope of a compile-time name encompasses all text subsequently scanned except those preprocessor procedures that have redeclared that identifier. The scope of a preprocessor variable that has been declared in a preprocessor procedure is the entire procedure (there is no nesting of preprocessor procedures).

When a preprocessor variable has been given a value, that value replaces all occurrences of the corresponding identifier in text other than preprocessor statements during the time that the variable is active. If the preprocessor variable is inactive (or if it has no value), replacement activity cannot occur for the corresponding identifier.

A preprocessor variable is activated initially by its appearance in the %DECLARE statement. It can be deactivated and subsequently reactivated by its appearance in %DEACTIVATE and %ACTIVATE statements, respectively. Deactivation of a preprocessor variable does not strip it of its value; in other words, an inactive preprocessor variable retains the value it had while it was active and can be altered by a preprocessor statement or procedure if so desired.

COMPILE-TIME EXPRESSIONS

Compile-time expressions are written and evaluated in the same way as source program expressions, with the following exceptions:

1. The operands of a compile-time expression can consist only of compile-time variables, references to compile-time procedures, decimal integer constants, bit-string constants, character-string constants, and references to the built-in function SUBSTR. Repetition factors are not allowed with the string constants and the arguments of a reference to SUBSTR must be compile-time expressions.

2. The exponentiation symbol (**) cannot be used.

3. For arithmetic operations, only decimal integer arithmetic of implementation-defined precision is performed. Note that the properties of the division operator are affected. For example, the expression 3/5 evaluates to 0, rather than to 0.6.

A character string in an expression being assigned to a compile-time variable may include compile-time variables, references to compile-time procedures, constants, and operators; preprocessor statements cannot be included in such strings.

COMPILE-TIME PROCEDURES

A compile-time procedure is a procedure that can be invoked only at the preprocessor stage. Its syntax differs from other PL/I procedures mainly in that its PROCEDURE and END statements must each have a leading percent symbol.

General format:

```
 %  label   :    [label:]...     PROCEDURE
 [(identifier[,identifier]...)]
 {CHARACTER|FIXED};
           .
           .
           .
[label:] RETURN (proprocessor-expression);
           .
           .
           .
% [label:] END [label];
```

Each identifier in the procedure statement is a parameter of the procedure; each parameter must be explicitly declared as FIXED or CHARACTER.

The label after the keyword END must be one of the labels of the procedure statement.

The CHARACTER or FIXED attribute in the compile-time procedure statement specifies the attribute to which the returned value is to be converted.

A compile-time procedure can be invoked only by a function reference. Recursive invocation of a compile-time procedure is not allowed, but the returned value, upon rescanning, can invoke the same procedure. Control cannot be transferred out of the procedure by a GO TO statement; consequently a RETURN statement must be executed to return both control, and the returned value, to the point of invocation.

The only statements and groups, besides one or more RETURN statements, that a compile-time procedure can contain are:

The null statement

The DECLARE statement

The assignment statement

The GO TO statement

The IF statement

The DO group

The syntax of these statements, and of the DO group is described under "Compile-time Statements" in this chapter, however, within a compile-time procedure, these statements must be written without percent symbols.

Names declared in a compile-time procedure are not known outside the procedure. Names declared in source text are known within the procedure unless they have been redeclared.

SCANNING COMPILE-TIME PROCEDURES AND FUNCTION REFERENCES

When the scan encounters a compile-time procedure, the procedure is skipped and scanning recommences after the END statement of the procedure.

If the scan is to recognise an identifier, with any required argument list, as a compile-time function reference the identifier must be declared in a compile-time DECLARE-statement as an entry name. The declaration of the entry name and the compile-time procedure must both be in source text; that is, any necessary INCLUDES must have been executed. The declaration, but not necessarily the procedure, must have been scanned. If the reference is not in a compile-time statement, the identifier must be activated; if it is not activated the identifier becomes part of the program text and the scan continues.

The argument list in a compile-time function reference is delimited by a balanced pair of parentheses whose left parenthesis is adjacent to the entry name, or is separated from the entry name by blanks and comments only. Commas which are not within further balanced parentheses separate the arguments from each other.

INVOCATION OF COMPILE-TIME PROCEDURES

The number of arguments in a compile-time function reference must be the same as the number specified in the entry attribute for the function's entry name; furthermore, the number of parameters specified in the entry attribute must be the same as the number indicated in the corresponding procedure statement.

The attributes of those parameters specified in the entry attribute must be the same as those declared for the corresponding parameters. For each parameter whose attributes are not specified in the entry attribute, the corresponding argument must have attributes identical to those declared for the parameters.

A compile-time function reference behaves differently when encountered in source text from when it is encountered in a compile-time statement. In source text the arguments are pieces of source text and the result returned by the function becomes part of source text; in a compile-time statement the arguments are compile-time expressions and the function returns a value for use in a compile-time expression.

In source text the arguments are delimited by the parentheses of the argument list, and by intervening commas in the text which are not themselves between balanced parentheses. The string of source text corresponding to each argument position is scanned and any necessary replacement is performed; the resulting sequence of characters is treated as an argument. If it is specified by an entry attribute, the argument will be converted to FIXED, otherwise it is treated as a character string. The value returned by the function is either FIXED or CHARACTER. In the former case the value is converted to CHARACTER and inserted in source text. In the latter case the string returned is scanned, and any necessary replacement takes place, before it is inserted in source text. Dummy arguments are always created when a function is invoked from source text.

In a compile-time statement the arguments of the function are compile time expressions; they are evaluated and any conversions specified by the entry attribute are performed before the function is invoked. Dummy arguments will be created where an argument is a constant, an expression in parentheses or where the attribute of the argument differs from that specified for the corresponding parameter in an entry attribute.

THE COMPILE-TIME BUILT-IN FUNCTION SUBSTR

The built-in function SUBSTR is the only built-in function that can be invoked during the preprocessor stage. It may be invoked from source text or from a compile-time statement.

The identifier SUBSTR is recognized as the built-in function name when it is encountered in a compile-time statement. However, if the identifier SUDSTR has already been declared as a variable or an entry name, a reference to SUBSTR in a compile-time statement is taken as a reference to the user-declared SUBSTR. When a programmer-written procedure named SUBSTR is in source text it is an error for the scan to encounter a reference to SUBSTR in a compile-time statement if no declaration for SUBSTR has been scanned. If the identifier SUBSTR refers to the built-in function, SUBSTR, it can be activated only by an ACTIVATE statement.

The built-in function SUBSTR behaves the same as a user compile-time function when encountered in source text or in a compile-time statement. The first argument is, if necessary, converted to character; the second and third arguments are, if necessary, converted to decimal. The returned value is a character string.

COMPILE-TIME STATEMENTS

Note that wherever keywords are shown below, they may be abbreviated as shown in Appendix 4. Note also that a comment appearing within a compile-time statement is never written into the program text.

THE ACTIVATE AND DEACTIVATE STATEMENTS

Function:

The appearance of an identifier in an ACTIVATE statement makes it eligible for replacement when certain conditions are met (see General Rules below); such an appearance is said to activate an identifier. The DEACTIVATE statement deactivates an identifier; that is, any subsequent appearance of such an identifier in the source program causes no replacement action (unless, of course, the identifier is again activated); the identifier remains unchanged.

General format:

```
% [label:] ... {ACTIVATE|DEACTIVATE} iden-
        tifier [,identifier] ...;
```

General rules:

1. Compile-time identifiers representing variables, procedure references, and the built-in function SUBSTR may be activated or deactivated.

2. When an identifier is deactivated, its appearance in the source program does

not cause any replacement action; the identifier is placed unchanged into the program text. However, any value that the identifier may have had before it was deactivated remains in effect as far as compile-time statements are concerned; deactivating an identifier only nullifies its ability to effect replacement.

3. When an identifier is activated, the following conditions must be met in order for replacement to occur:

   a. The identifier must not appear within a comment or a character string.

   b. The identifier must be immediately preceded and followed by a PL/I delimiter.

   If both conditions are met, the replacement value for the compile-time variable or procedure reference is converted to a character string and then placed into the program text (assuming that the rescan does not cause any further replacement). A single blank is inserted immediately preceding and following the value.

Note: The appearance of an identifier in a DECLARE statement serves to activate that identifier initially. Therefore, an identifier need be activated by an ACTIVATE statement only if it has been explicitly deactivated.

Example:

If the source text contains the following statements:

```
    % DECLARE I FIXED, T CHARACTER;
    % DEACTIVATE I;
    % I = 15;
    % T = 'A(I)';
 S = I*T*3;
    % I = I + 5;
    % ACTIVATE I;
    % DEACTIVATE T;
 R = I*T*2;
```

then the program text generated by the above would be:

```
    S = I* A(I) *3;
    R =  20 *T*2;
```

THE ASSIGNMENT STATEMENT

Function:

The compile-time assignment statement is used to evaluate compile-time expressions and to assign the result to a compile-time variable.

General format:

```
% [label:] ...  compile-time-variable =
      compile-time-expression;
```

General rules:

1. For arithmetic operations, only decimal integer arithmetic of precision (p,0) is performed (p is implementation-defined); that is, each operand is converted to a decimal fixed-point value of precision (p,0) before the operation is performed, and the decimal fixed-point result is converted to precision (p,0) also. Any character string being converted to an arithmetic value must be in the form of an optionally signed decimal integer constant.

2. The conversion of a fixed-point decimal quantity to a character-string always results in a string of length p+3.

3. The value assigned to a compile-time character-string variable may include percent signs, unmatched quotation marks, and unmatched comment delimiters.

THE DECLARE STATEMENT

Function:

The DECLARE statement establishes an identifier as a compile-time variable or a compile-time procedure name. The appearance of an identifier in a compile-time DECLARE statement activates that identifier; that is, it indicates to the preprocessor that this identifier may cause replacement action in the source program.

General format:

```
% [label:]... DECLARE identifier
              attribute-list [,identifier
              attribute-list]...;
```

where "attribute list" is defined as:

```
CHARACTER|FIXED|ENTRY[([CHARACTER|FIXED]
  [,[CHARACTER|FIXED]]...)]
  RETURNS (CHARACTER|FIXED)
```

Syntax rules:

1. The attributes may be factored as in PL/I source program DECLARE statements.

2. Although the DECLARE statement may be labeled, all such labels are ignored.

General rules:

1. A length cannot be specified with the CHARACTER attribute. If CHARACTER is specified, it is assumed that the associated identifier represents a varying character string that has no maximum length.

2. A compile-time declaration is not known until it has been scanned by the preprocessor. Any reference to a compile-time variable or compile-time procedure name encountered in a compile-time statement before the variable or procedure name has been declared is in error.

3. The scope of a compile-time variable name, compile-time procedure name, or a label of a compile-time statement is the entire text scanned by the processor, not including any compile-time procedures that redeclare the identifier. The scope of a name declared in a compile-time procedure is limited to that procedure.

4. Multiple declaration of compile-time variables or labels are not allowed.

5. A compile-time DECLARE statement is executed only the first time it is encountered; any subsequent scanning through the statement has no effect.

THE DO STATEMENT

General format:

```
% [label:] ...  DO[i = m1 ┌TO m2 [BY m3]┐ ];
                         └BY m3[TO m2] ┘
         .
         .
         .
% [label:] ...  END [label];
```

Syntax rule:

The <u>i</u> represents a compile-time variable, and m1, m2, and m3 are compile-time expressions.

General rules:

1. Transfer may not be made into an iterative DO group except via a return from a compile-time procedure invoked from within the group.

2. The text of a DO group may consist of both compile-time statements and source program statements. The source program statements are not executed; they are scanned for possible replacement activity. Thus, the example below results in the same expansion generated by the example called "Compile-Time Loop Expansion" in the section "Rescanning and Replacement."

```
% DECLARE I FIXED;
% DO I = 1 TO 10;
Z(I) = X(I) + Y(I);
% END;
% DEACTIVATE I;
```

3. The expansion of the DO is the same as for source program DO groups, with the PL/I source program statements replaced by the equivalent compile-time statements.

THE GO TO STATEMENT

Function:

The compile-time GO TO statement causes the processor to resume its scan at the specified label.

General format:

```
% [label:] ...  {GO TO|GOTO} label;
```

General rule:

1. The label that determines the point at which the scan will resume must be the label of a compile-time statement.

2. A compile-time GO TO statement can be used to transfer control from included text to a compile-time statement in the containing text, but the reverse is in error.

THE IF STATEMENT

Function:

The compile-time IF statement controls the flow of the processor's scan according to the value of a compile-time expression.

General format:

```
% [label:] ... IF compile-time-expression
         % THEN compile-time-group-1
         [% ELSE compile-time-group-2]
```

Syntax rule:

A compile-time group is any single executable compile-time statement or a compile-time DO group (see below).

General rules:

1. The compile-time expression is evaluated and converted to a bit string. (If the conversion cannot be made, it is an error.) If any bit in the string has the value 1, compile-time group-1 is executed and group-2, if present, is skipped. Otherwise, group-1 is skipped and group-2, if present, is executed. In either case, the scan resumes immediately following the IF statement, unless, of course, a compile-time GO TO statement in one of the groups has caused the processor to resume its scan elsewhere.
2. Compile-time IF statements may be nested. See General Rule 2 of "The IF Statement," Chapter 8.

THE INCLUDE STATEMENT

Function:

The INCLUDE statement is used to incorporate strings of external text into the program text being formed.

General format:

```
%[label:] ... INCLUDE text-identification
     [,text-identification]...;
```

where "text-identification" is of the form:

$$\left\{\begin{array}{l}\text{identifier-1 [(identifier-2)]}\\ \text{[identifier-1] (identifier-2)}\end{array}\right\}$$

General rules:

1. Each text identification is used in an implementation-defined manner to identify a data set. This data set may contain source program text and/or compile-time statements.
2. The incorporated data sets are scanned, in sequence, in the same manner as the source text, i.e., replacements are made and compile-time statements are executed. Thus, they may contribute to the final program text. Note that the included text does not replace the INCLUDE statement, which is executed again if it is reencountered in the scan.
3. A transfer of control from included text to a statement in the containing text is valid, but the reverse is in error. (Note that "transfer of control" should be taken in the sense of a GO TO statement only; a "transfer of control" in the sense of invoking a compile-time procedure is always permissible.)
4. Compile-time IF statements, DO groups, and procedures must each be complete within a single included data set.

Examples:

1. Assume that the data set named PAYRL contains the following structure declaration:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHARACTER (30) VARYING,
            3 FIRST CHARACTER (15) VARYING,
            3 MIDDLE CHARACTER (3) VARYING,
          2 MAN_NO FIXED DECIMAL (6,0),
          2 HOURS,
            3 REGLR FIXED DECIMAL (8,2),
            3 OVRTIM FIXED DECIMAL (8,2),
          2 RATE,
            3 REGLAR FIXED DECIMAL (8,2),
            3 OVERTIME FIXED DECIMAL (8,2);
```

then the following compile-time program

```
% DECLARE PAYROLL CHARACTER;
% PAYROLL = 'CUM_PAY';
% INCLUDE PAYRL;
% DEACTIVATE PAYROLL;
% INCLUDE PAYRL;
```

would generate two identical structure descriptions in the program text, the only difference being their names, CUM_PAY and PAYROLL.

2. If the source text contained the following:

```
% DECLARE(FILENAME1,FILENAME2)
  CHARACTER;
  % FILENAME1 = 'MASTER';
  % FILENAME2 = 'NEWFILE';
  % INCLUDE DECLARATIONS;
```

and if the data set named DECLARATIONS contained

```
DECLARE
    FILENAME1 FILE RECORD INPUT
          DIRECT KEYED,
    FILENAME2 FILE RECORD OUTPUT
          DIRECT KEYED;
```

then the program text would contain the following statement:

```
DECLARE
    MASTER  FILE  RECORD  INPUT DIRECT
       KEYED,
    NEWFILE FILE RECORD OUTPUT  DIRECT
       KEYED;
```

Note that in this way a central library of file declarations can be used, with each user supplying his own names for the files being declared.

THE NULL STATEMENT

Function:

The compile-time null statement is used to insert compile-time labels into the text; these labels are transfer targets for compile-time GO TO statements.

General format:

```
% [label:] ...;
```

APPENDIX 1: BUILT-IN FUNCTIONS AND PSEUDO-VARIABLES

All of the built-in functions and pseudo-variables that are available to the PL/I programmer are given in this appendix, and are presented in alphabetical order under their respective headings. The general organization of the appendix is as follows:

1. Computational Built-in Functions
   a. String-handling built-in functions
   b. Arithmetic built-in functions
   c. Mathematical built-in functions
   d. Array manipulation built-in functions

2. Condition Built-in Functions

3. Based Storage Built-in Functions

4. Multitasking Built-in Functions

5. Miscellaneous Built-in Functions

6. Pseudo-Variables

The <u>computational built-in functions</u> provide string handling, arithmetic operations (addition, division, etc.), mathematical operations (trigonometric functions, square root, etc.), and array manipulation. The computational built-in functions are:

<u>String Handling</u>:

BIT	LOW
BOOL	REPEAT
CHAR	STRING
HIGH	SUBSTR
INDEX	UNSPEC
LENGTH	

<u>Arithmetic</u>:

ABS	IMAG
ADD	MAX
BINARY	MIN
CEIL	MOD
COMPLEX	MULTIPLY
CONJG	PRECISION
DECIMAL	REAL
DIVIDE	ROUND
FIXED	SIGN
FLOAT	TRUNC
FLOOR	

<u>Mathematical</u>:

ATAN	LOG10
ATAND	LOG2
ATANH	SIN
COS	SIND
COSD	SINH
COSH	SQRT
ERF	TAN
ERFC	TAND
EXP	TANH
LOG	

<u>Array Manipulation</u>:

ALL	LBOUND
ANY	POLY
DIM	PROD
HBOUND	SUM

The <u>condition built-in functions</u> allow the PL/I programmer to investigate interrupts arising from enabled ON-conditions. The condition built-in functions are:

DATAFIELD	ONFILE
ONCHAR	ONKEY
ONCODE	ONLOC
ONCOUNT	ONSOURCE

The <u>based storage built-in functions</u> are designed to facilitate list processing and the use of based storage. They mainly return special values to locator and area variables. The based storage built-in functions are:

ADDR	NULL	OFFSET
EMPTY	NULLO	POINTER

The <u>multitasking built-in functions</u> allow the programmer to investigate the current state of a task or asynchronous input/output operation, or the current priority of a task. The multitasking built-in functions are:

COMPLETION
PRIORITY
STATUS

The <u>miscellaneous built-in functions</u> perform various duties; for example, one function provides the current date, another provides a count of data items transmitted during a STREAM input/output operation, while another provides an indication of whether or not a controlled variable is in an allocated state. The miscellaneous built-in functions are:

ALLOCATION	LINENO
COUNT	TIME
DATE	

Each of the pseudo-variables is described briefly. A more detailed description can be found in the discussion of the corresponding built-in function. The pseudo-variables are:

COMPLETION	PRIORITY
COMPLEX	REAL
IMAG	STATUS
ONCHAR	SUBSTR
ONSOURCE	UNSPEC

COMPUTATIONAL BUILT-IN FUNCTIONS

STRING HANDLING BUILT-IN FUNCTIONS

The functions described in this section may be used for manipulating strings. Unless it is specifically stated otherwise, any argument can be a scalar or aggregate expression (see "Built-in Functions with Aggregate Arguments," in Chapter 3). An argument that is specified as "string" can be an expression of any data type, but if it is arithmetic, it is converted to bit-string (if binary base) or character-string (if decimal base) before the function is invoked.

Conversions are made according to the rules for data conversion.

BIT String Built-in Function

Definition: BIT converts a given value to a bit string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a bit-string conversion.

Reference: BIT (value [,length])

Arguments: The argument, "value," is an expression representing the quantity to be converted to a bit string. The argument, "length," when specified, is an expression whose value gives the length of the result. If "length" is not specified, it is determined according to the type conversion rules.

Result: The value returned by this function is "value" converted to a bit string. The length of this bit string is determined by the integral value of "length," as described above.

BOOL String Built-in Function

Definition: BOOL produces a bit string whose bit representation is a result of a given boolean operation on two given bit strings.

Reference: BOOL (x,y,w)

Arguments: Arguments "x" and "y" are the two bit strings upon which the boolean operation specified by "w" is to be performed. If "x" and "y" are not bit strings, they are converted to bit strings. If "x" and "y" differ in length, the shorter string is extended with zeros on the right to match the length of the longer string.

Argument "w" represents the boolean operation. It is a bit string of length 4 and is defined as n_1 n_2 n_3 n_4, where each _n_ is either 0 or 1. There are 16 possible bit combinations and thus 16 possible boolean operations. If necessary, "w" is converted to a bit string (of length 4) before the function is invoked, if necessary.

Result: The value returned by this function is a bit string, _z_, whose length is equal to the longer of "x" and "y." Each bit of _z_ is determined by the boolean operation on the corresponding bits of "x" and "y" as follows: the _i_th bit of _z_ is set to the value of n_1, n_2, n_3, or n_4 depending on the combination of the _i_th bits of "x" and "y" as shown in the boolean table below:

xi	yi	zi
0	0	n_1
0	1	n_2
1	0	n_3
1	1	n_4

CHAR String Built-in Function

Definition: CHAR converts a given value to a character string and returns the result to the point of invocation. This function allows the programmer to control the size of the result of a character-string conversion.

Reference: CHAR (value [,length])

Arguments: The argument, "value," is an expression representing the quantity to be converted to a character string. The argument, "length," when specified, is an expression whose integral value gives the length of the result. If "length" is not specified, it is determined according to the type conversion rules.

Result: The value returned by this function is "value" converted to a character string. The length of this character string is determined by the integral value of "length," as described above.

HIGH String Built-in Function

Definition: HIGH forms a character string of a given length from the highest character in the collating sequence; that is, each character in the constructed string is the highest character in the collating sequence.

Reference: HIGH (length)

Argument: The argument, "length," is an expression whose integral value specifies the length of the string that is to be formed.

Result: The value returned by this function is a character string whose length is determined by the integral value of "length"; each character in the string is the highest character in the collating sequence.

INDEX String Built-in Function

Definition: INDEX searches a specified string for a specified bit or character string configuration. If the configuration is found, the starting location of that configuration within the string is returned to the point of invocation.

Reference: INDEX (string, config)

Arguments: Two arguments must be specified. The first argument, "string," represents the string to be searched; the second argument, "config," represents the bit or character string configuration for which "string" is to be searched. If neither argument is a bit string, or if only one argument is a bit string, both arguments are converted to character strings. If both arguments are bit-string, no conversion is performed.

Result: The value returned by this function is a binary integer of default precision. This binary integer is either:

1. The location in "string" at which "config" has been found. If more than one "config" exists in "string," the location of the first one found (in a left-to-right sense) will be returned.

2. The value 0, if "config" does not exist within "string" or if either of the arguments has a length of zero.

LENGTH String Built-in Function

Definition: LENGTH finds the string length of a given value and returns it to the point of invocation.

Reference: LENGTH (string)

Argument: The argument, "string," represents a character string or a bit string whose length is to be found. The argument need not represent a string; if it does not, it is converted before the function is invoked to a character string (if the argument is DECIMAL) or a bit string (if the argument is BINARY).

Result: The value returned by this function is a fixed binary integer of default precision giving the current length of "string." If "string" is an array expression, an array of identical bounds is returned.

Example: If XYZ is a varying-length character string whose maximum length is 30, but whose current length is 25, then the statement:

```
     I = LENGTH(SUBSTR(XYZ,4));
```

will assign a binary value of 22 to I.

LOW String Built-in Function

Definition: LOW forms a character string of specified length from the lowest character in the collating sequence; i.e., each character of the formed string will be the lowest character in the collating sequence.

Reference: LOW (length)

Argument: The argument, "length," is an expression whose integral value specifies the length of the string being formed.

Result: The value returned by this function is a character string whose length is determined by the integral value of "length"; each character in the string is the lowest character in the collating sequence.

REPEAT String Built-in Function

Definition: REPEAT takes a given string value and forms a new string consisting of the given string value concatenated with itself a specified number of times.

Reference: REPEAT (string,factor)

Arguments: The argument "string" represents the string from which the new string will be formed. If this argument is not a string, it will be converted to a string.

The argument "factor" is an expression whose integral value specifies the number of times that "string" is to be concatenated with itself; "factor" can be signed.

Result: The value returned by this function is "string" concatenated with itself "factor" times. In other words, the returned value will be a string containing (factor+1) occurrences of the value "string." If "factor" is less than or equal to zero, the returned value is identical to the argument (i.e., the converted argument, if the original argument was not a string).

STRING String Built-in Function

Definition: STRING concatenates all the elements in the result of an expression into a single string element.

Reference: STRING (x)

Arguments: The argument, "x," is an element, array, or structure expression, whose result is composed either entirely of character strings and/or decimal numeric character data, or entirely of bit strings and/or binary numeric character data.

Result: The value returned by this function is an element bit string or character string, the concatenation of all the elements in "x." If "x" contains one or more varying strings, the result is a varying string.

SUBSTR String Built-in Function

Definition: SUBSTR extracts a substring of user-defined length from a given string and returns the substring to the point of invocation. (SUBSTR can also be used as a pseudo-variable.)

Reference: SUBSTR (string,i[,j])

Arguments: The argument "string" represents the string from which a substring will be extracted. If this argument is not a string, it will be converted to a string. Argument "i" represents the starting point of the substring and "j" represents the length of the substring. Arguments "i" and "j" must be integers or expressions that can be converted to integers.

Assuming that the length of "string" is *k*, arguments "i" and "j" must satisfy the following conditions:

1. j must be less than or equal to k and greater than or equal to 0.

2. i must be less than or equal to k and greater than or equal to 1.

3. The value of i+j-1 must be less than or equal to k.

Thus, the substring, as specified by "i" and "j" must lie within "string."

If "j" is not specified, it is assumed to be equal to the value of k-i+1. In other words, it is assumed to be the length of the remainder of "string," beginning at the *i*th position in "string."

When these conditions are not satisfied, the SUBSTR reference causes the STRINGRANGE interrupt to be raised, if it is enabled.

Result: The value returned by this function is a varying-length string whose current length is defined as follows:

1. If j=0, the returned value is the null string.

2. If j is greater than 0, the returned value is that substring beginning at the *i*th character or bit of the first argument and extending j characters or bits.

3. If j is not specified, the returned value is that substring beginning at the *i*th character or bit and extending to the end of "string."

UNSPEC String Built-in Function

Definition: UNSPEC returns a bit string that is the internal coded representation of a given value. (UNSPEC can also be used as a pseudo-variable.)

Reference: UNSPEC (x)

Argument: The argument, "x," may be an arithmetic, string, locator, or area expression, or an area variable, whose internal coded representation is to be found.

Result: The value returned by this function is the internal coded representation of "x" and is implementation defined. This representation is in bit-string form. The length of this string depends upon the attributes of "x".

ARITHMETIC BUILT-IN FUNCTIONS

All values returned by arithmetic built-in functions are in coded arithmetic form. The arguments of these functions should also be in that form. If an argument is not coded arithmetic, then, before the function is invoked, it is converted to coded arithmetic according to the rules for data conversion. Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In some function descriptions, the phrase "converted to the highest characteristics" is used; this means that the rules for mixed characteristics are followed. See "Mixed Characteristics", Chapter 3.

In general, an argument of an arithmetic built-in function may be a scalar or aggregate expression (see "Built-in Functions with Aggregate Arguments," in Chapter 3).

Unless it is specifically stated otherwise:

1. The mode of an argument may be real or complex.

2. The base, scale, mode, and precision of the returned value are determined according to the rules for the conversion of expression operands.

In many of these built-in function descriptions, the symbol N is used. This symbol represents the maximum precision permitted by an implementation for the given base and scale.

ABS Arithmetic Built-in Function

Definition: ABS finds the absolute value of a given quantity and returns it to the point of invocation.

Reference: ABS (x)

Argument: The quantity whose absolute value is to be found is given by "x."

Result: The value returned by this function is the absolute value of "x." If "x" is real, the result is the positive value of "x"; if "x" is complex, the result is the positive square root of the sum of the squares of the real and imaginary parts of "x." The mode of the result is real, while the base, scale, and precision are the same as those of "x," with one exception: if "x" is a complex fixed-point value of precision (p,q), the precision of the result is:

(MIN(N,p+1),q)

ADD Arithmetic Built-in Function

Definition: ADD finds the sum of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of an add operation.

Reference: ADD (x,y,p[,q])

Arguments: Arguments "x" and "y" represent the values to be added. Arguments "p" and "q" must be decimal integer constants specifying the precision of the result; "q" may be signed. If the scale of the result is fixed-point, both "p" and "q" must be specified; if the scale of the result is floating-point, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the sum of "x" and "y." The precision of the result is determined by "p" and "q"; this precision is maintained throughout the execution of the function.

BINARY Arithmetic Built-in Function

Definition: BINARY converts a given value to binary base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a binary conversion.

Reference: BINARY (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to binary base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the binary result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, the precision of the result is determined according to the standard rules for data conversion. Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the binary equivalent of "x." The scale and mode of this value are the same as those of "x." The precision is given by "p" and "q."

CEIL Arithmetic Built-in Function

Definition: CEIL determines the smallest integer that is greater than or equal to a given real value and returns that integer to the point of invocation.

Reference: CEIL (x)

Argument: The argument, "x," must not be complex.

Result: The value returned by this function is the smallest integer that is greater than or equal to "x." The base, scale, mode, and precision are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is defined as:

```
(MIN(N,MAX(p-q+1,1)),0)
```

COMPLEX Arithmetic Built-in Function

Definition: COMPLEX forms a complex number from two given real values and returns it to the point of invocation. (COMPLEX can also be used as a pseudo-variable.)

Reference: COMPLEX (x,y)

Arguments: Arguments "x" and "y" must both be real; "x" represents the real part of the complex number to be formed and "y" represents the imaginary part.

Result: The value returned by this function is the complex number that has been formed from "x" and "y."

CONJG Arithmetic Built-in Function

Definition: CONJG finds the conjugate of a complex value and returns it to the point of invocation. (The conjugate of a complex number is the complex number with the sign of the imaginary part reversed.)

Reference: CONJG (x)

Argument: The argument, "x," is the value whose conjugate is to be found; it must be complex.

Result: The value returned by this function is the conjugate of "x." The base, scale, mode, and precision of the conjugate are the same as those of the argument.

DECIMAL Arithmetic Built-in Function

Definition: DECIMAL converts a given value to decimal base and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a decimal conversion.

Reference: DECIMAL (x[,p[,q]])

Arguments: The first argument, "x," represents the value to be converted to decimal base. Arguments "p" and "q," when specified, must be decimal integer constants giving the precision of the decimal result; "q" may be signed. The precision of a fixed-point result is (p,q); the precision of a floating-point result is (p). If both "p" and "q" are omitted, however, the precision of the result is determined according to the standard rules for data conversion. Note that "q" must be omitted for floating-point arguments.

Result: The value returned by this function is the decimal equivalent of the argument "x." The scale and mode of this value are the same as argument "x"; its precision is given by "p" and "q."

DIVIDE Arithmetic Built-in Function

Definition: DIVIDE divides a given value by another given value and returns the quotient to the point of invocation. This function allows the programmer to control the precision of the result of a divide operation.

Reference: DIVIDE (x,y,p[,q])

Arguments: The argument, "x," is the dividend and argument "y" is the divisor. Arguments "p" and "q" ("q" is optional and may be signed) must be decimal integer constants specifying the precision of the result. If the result is a fixed-point value, "p" and "q" must both be specified; if the result is a floating-point value, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the quotient resulting from the division of "x" by "y." The precision of the result is determined by "p" and "q" as described above; this precision is maintained throughout the execution of the function.

FIXED Arithmetic Built-in Function

Definition: FIXED converts a given value to fixed-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a fixed-point conversion.

Reference: FIXED (x[,p[,q]])

Argument: The first argument, "x," represents the value to be converted to fixed-point scale. Arguments "p" and "q," when specified, must be decimal integer constants ("q" can be signed) giving the precision of the result, (p,q). If "q" is omitted, zero is assumed. If both "p" and "q" are omitted, precision of the result will be default fixed-point precision for the base of "x."

Result: The value returned by this function is the fixed-point equivalent of the argument "x"; its precision is (p,q).

FLOAT Arithmetic Built-in Function

Definition: FLOAT converts a given value to floating-point scale and returns the converted value to the point of invocation. This function allows the programmer to control the precision of the result of a floating-point conversion.

Reference: FLOAT (x[,p])

Arguments: The first argument, "x," represents the value to be converted to floating-point scale. The second argument, "p," when specified, must be a decimal integer constant giving the precision of the result. If "p" is omitted, precision of the result will be floating-point default precision for the base of "x."

Result: The value returned by this function is the floating-point equivalent of "x"; its precision is "p."

FLOOR Arithmetic Built-in Function

Definition: FLOOR determines the largest integer that does not exceed a given value and returns that integer to the point of invocation.

Reference: FLOOR (x)

Argument: The argument, "x," must not be complex.

Result: The value returned by this function is the largest integer that does not exceed "x." The base, scale, mode, and precision of this value are the same as those of "x," with one exception: if "x" is a fixed-point value of precision (p,q), the precision of the result is:

```
(MIN(N,MAX(p-q+1,1)),0)
```

IMAG Arithmetic Built-in Function

Definition: IMAG extracts the imaginary part of a given complex number and returns it to the point of invocation. (IMAG can also be used as a pseudo-variable.)

Reference: IMAG (x)

Argument: The argument, "x," is the complex value whose imaginary part is to be extracted.

Result: The value returned by this function is the imaginary part of "x." The base, scale, and precision of the imaginary part are unchanged. The mode of the returned value is real.

MAX Arithmetic Built-in Function

Definition: MAX extracts the highest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MAX (x_1,x_2,...,x_n)

Arguments: Two or more arguments must be given. The arguments must not be complex.

Result: The value returned by MAX is the value of the maximum-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

$(p_1,q_1), (p_2q_2),\ldots, (p_n,q_n)$

then the precision of the result is as follows:

$$(\mathrm{MIN}(N,\mathrm{MAX}(p_1-q_1,\ldots,p_n-q_n)+ \mathrm{MAX}(q_1,\ldots,q_n)),\mathrm{MAX}(q_1,\ldots q_n))$$

MIN Arithmetic Built-in Function

Definition: MIN extracts the lowest-valued expression from a given set of two or more expressions and returns that value to the point of invocation.

Reference: MIN $(x_1,x_2,\ldots,x_n)$

Arguments: Two or more arguments must be given. The arguments must not be complex.

Result: The value returned by MIN is the value of the lowest-valued argument. The returned value is converted to conform to the highest characteristics of all the arguments that were specified. If the arguments are fixed-point values and have precisions:

$(p_1,q_1), (p_2,q_2),\ldots,(p_n,q_n)$

then the precision of the result is as follows:

$$(\mathrm{MIN}(N,\mathrm{MAX}(p_1-q_1,\ldots,p_n-q_n)+ \mathrm{MAX}(q_1,\ldots q_n)),\mathrm{MAX}(q_1,\ldots,q_n))$$

MOD Arithmetic Built-in Function

Definition: MOD extracts the remainder resulting from the division of one real quantity by another and returns it to the point of invocation.

Reference: MOD (x_1,x_2)

Arguments: Two arguments must be given. They must not be complex. Before the function is invoked, the base and scale of each argument are converted according to the standard rules for data conversion.

Result: The value returned by MOD is the positive remainder resulting from the division of "x_1" by "x_2." If the result is in floating-point scale, its precision is the higher of the precisions of the arguments; if the result is in fixed-point scale, its precision is defined as follows:

$$(\mathrm{MIN}(N,p_2-q_2+\mathrm{MAX}(q_1,q_2)),\mathrm{MAX}(q_1,q_2))$$

where (p_1,q_1) and (p_2,q_2) are the precision of "x_1" and "x_2," respectively.

MULTIPLY Arithmetic Built-in Function

Definition: MULTIPLY finds the product of two given values and returns it to the point of invocation. This function allows the programmer to control the precision of the result of a multiplication operation.

Reference: MULTIPLY $(x_1,x_2,p[,q])$

Arguments: Arguments "x_1" and "x_2" represent the values to be multiplied. Arguments "p" and "q" ("q" is optional and may be signed) are decimal integer constants specifying the precision of the result. If the result is a fixed-point value, "p" and "q" must both be specified; if the result is a floating-point value, only "p" must be specified. In either case, "p" must not exceed N.

Result: The value returned by this function is the product of "x_1" and "x_2." The precision of the result is as specified; this precision is maintained throughout the execution of the function.

PRECISION Arithmetic Built-in Function

Definition: PRECISION converts a given value to a specified precision and returns the converted value to the point of invocation.

Reference: PRECISION (x,p[,q])

Arguments: The first argument, "x," represents the value to be converted to the specified precision. Arguments "p" and "q" ("q" is optional and may be signed) are decimal integer constants specifying the precision of the result. If "x" is a fixed-point value, "p" and "q" must be specified; if "x" is a floating-point value, only "p" must be specified.

Result: The value returned by this function is the value of "x" converted to the specified precision. The base, scale, and mode of the returned value are the same as those of "x."

REAL Arithmetic Built-in Function

Definition: REAL extracts the real part of a given complex value and returns it to the point of invocation. (REAL can also be used as a pseudo-variable.)

Reference: REAL (x)

Argument: The argument, "x," must be a complex expression.

Result: The value returned by this function is the real part of the complex value represented by "x." The base, scale, and precision of the real part are unchanged.

ROUND Arithmetic Built-in Function

Definition: ROUND rounds a given value at a specified digit and returns the rounded value to the point of invocation.

Reference: ROUND (expression,n)

Arguments: The first argument, "expression," is an element or array representing the value (or values, in the case of an array expression) to be rounded; the second argument, "n," is a signed or unsigned decimal integer constant specifying the digit at which the value of "expression" is to be rounded. If "n" is positive, rounding occurs at the nth digit to the right of the decimal (or binary) point in the value of "expression"; if "n" is negative, rounding occurs at the nth digit to the left of decimal (or binary) point in the value of "expression." Note that the decimal (or binary) point is assumed to be at the left for floating-point values.

Result: For fixed-point values, ROUND returns the value of "expression" rounded at the nth digit to the right of the decimal (or binary) point for positive "n" or to the left of the decimal (or binary) point for negative "n." Thus, when "n" is negative, the returned value is an integer.

If "expression" is a floating-point expression, the second argument is ignored, and the rightmost bit in the internal floating-point representation of the expression's value is set to 1 if it is 0. If the rightmost bit is 1, it is left unchanged.

If "expression" is a string, the returned value is the same string unmodified.

The base, scale, mode, and precision of the returned value are those of the value of "expression," with one exception: if the value of "expression" is fixed-point of precision (p,q), the result is fixed-point of precision:

```
(MIN(p+1,N),q)
```

Note that the rounding of a negative quantity results in the rounding of the absolute value of that quantity.

SIGN Arithmetic Built-in Function

Definition: SIGN determines whether a value is positive, negative, or zero, and it returns an indication to the point of invocation.

Reference: SIGN (x)

Argument: The argument, "x," must not be complex.

Result: This function returns a real fixed-point binary value of default precision according to the following rules:

1. If the argument is greater than 0, the returned value is 1.
2. If the argument is equal to zero, the returned value is 0.
3. If the argument is less than zero, the returned value is -1.

TRUNC Arithmetic Built-in Function

Definition: TRUNC truncates a given value to an integer as follows: first, it determines whether a given value is positive, negative, or equal to zero. If the value is negative, TRUNC returns the smallest integer that is greater than that value; if the value is positive or equal to zero, TRUNC returns the largest integer that does not exceed that value.

Reference: TRUNC (x)

Argument: The argument, "x," must not be complex.

Result: If "x" is less than zero, the value returned by TRUNC is CEIL(x). If "x" is greater than or equal to zero, the value returned by TRUNC is FLOOR(x). In either case, the base, scale, and mode of the result are the same as those of "x." If "x" is a floating-point value, the precision remains the same. If "x" is a fixed-point value of precision (p,q), the precision of the result is:

```
    (MIN(N,MAX(p-q+1,1)),0)
```

MATHEMATICAL BUILT-IN FUNCTIONS

All arguments to the mathematical built-in functions should be in coded arithmetic form and in floating-point scale. Any argument that does not conform to this rule is converted to coded arithmetic and floating-point before the function is invoked, according to the standard rules for data conversion. Note, therefore, that in the function descriptions below, a reference to an argument always means the converted argument, if conversion was necessary.

In general, an argument to a mathematical built-in function may be a scalar or aggregate expression (see "Built-in Functions with Aggregate Arguments," in Chapter 3).

Unless it is specifically stated otherwise, an argument may be real or complex. Tables 3 and 4 at the end of this section provide a quick reference for those mathematical functions that accept either real or complex arguments and those that accept only real arguments.

All of the mathematical built-in functions return coded arithmetic floating-point values. The mode, base, and precision of these values are always the same as those of the arguments.

ATAN Mathematical Built-in Function

Definition: ATAN finds the arctangent of a given value and returns the result expressed in radians, to the point of invocation.

Reference: ATAN (x[,y])

Arguments: The argument, "x," must always be specified; the argument "y" is optional. If "y" is omitted, "x" represents the value whose arctangent is to be found; in such a case, "x" may be real or complex, but if it is complex, it must not be equal to ±1i.

If "y" is specified, then the value whose arctangent is to be found is taken to be the expression x/y. In this case, both "x" and "y" must be real, and both cannot be equal to 0 at the same time.

Result: When "x" alone is specified, the value returned by ATAN depends on the mode of "x." If "x" is real, the returned value is the arctangent of "x," expressed in radians, where:

```
    -pi/2<ATAN(x)<pi/2
```

If "x" is complex, the arctangent function is multiple-valued, and hence only the principal value can be returned. The principal value of ATAN for a complex argument "x" is defined as follows:

```
    -i*ATANH(i*x)
```

If both "x" and "y" are specified, the possible values returned by this function are defined as follows:

1. If y>0, the value is arctangent (x/y) in radians.
2. If x>0 and y=0, the value is (pi/2) radians.
3. If x≥0 and y<0, the value is (pi+ arctangent (x/y)) radians.
4. If x<0 and y=0, the value is (-pi/2) radians.
5. If x<0 and y<0, the value is (-pi+ arctangent (x/y)) radians.

ATAND Mathematical Built-in Function

Definition: ATAND finds the arctangent of a given real value and returns the result, expressed in degrees, to the point of invocation.

Reference: ATAND (x[,y])

Arguments: Arguments "x" and "y" ("y" may be omitted) must be real values. If "y" is omitted, "x" represents the value whose arctangent is to be found. If "y" is specified, the value whose arctangent is to be found is represented by the expression x/y; in this case, both "x" and "y" cannot be equal to 0 at the same time.

Result: If "y" is not specified, the value returned by this function is simply the arctangent of "x," expressed in degrees, where:

```
    -90<ATAND(x)<90
```

If "y" is specified, the value returned by this function is ATAN (x,y), except that the value is expressed in degrees and not in radians (see "ATAN Mathematical Built-in Function" in this section); that is, the returned value is defined as:

ATAN(x,y) = (180/pi)*ATAN(x,y)

ATANH Mathematical Built-in Function

Definition: ATANH finds the inverse hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: ATANH (x)

Argument: The value whose inverse hyperbolic tangent is to be found is represented by "x." If "x" is real, the absolute value of "x" must not be greater than or equal to 1; that is, for real "x," it is an error if ABS(x)≥1. If "x" is complex, it must not be equal to ±1.

Result: If "x" is real, the value returned by this function is the inverse hyperbolic tangent of "x." For complex "x," the inverse hyperbolic tangent is defined as follows:

(LOG((1+x)/(1-x)))/2

COS Mathematical Built-in Function

Definition: COS finds the cosine of a given value, which is expressed in radians, and returns the result to the point of invocation.

Reference: COS (x)

Argument: The value whose cosine is to be found is given by "x"; this value can be real or complex and must be expressed in radians.

Result: The value returned by this function is the cosine of "x." For complex argument "x," the cosine of "x" is defined below, where x = y_1+iy_2:

$$\cos(x)=\cos(y_1)*\cosh(y_2)-i*\sin(y_1)*\sinh(y_2)$$

COSD Mathematical Built-in Function

Definition: COSD finds the cosine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: COSD (x)

Argument: The value whose cosine is to be found is given by "x"; this value must be real and must be expressed in degrees.

Result: The value returned by this function is the cosine of "x."

COSH Mathematical Built-in Function

Definition: COSH finds the hyperbolic cosine of a given value and returns the result to the point of invocation.

Reference: COSH (x)

Argument: The value whose hyperbolic cosine is to be found is given by "x."

Result: The value returned by this function is the hyperbolic cosine of "x." For complex argument "x," the hyperbolic cosine of "x" is defined below, where x = y_1+iy_2:

$$\cosh(x)=\cosh(y_1)*\cos(y_2)+i*\sinh(y_1)*\sin(y_2)$$

ERF Mathematical Built-in Function

Definition: ERF finds the error function of a given real value and returns it to the point of invocation.

Reference: ERF (x)

Argument: The value for which the error function is to be found is represented by "x"; this value must be real.

Result: The value returned by this function is defined as follows:

$$\mathrm{ERF}(x)= \frac{2}{\sqrt{\Pi}}\int_0^x e^{-t^2}dt$$

ERFC Mathematical Built-in Function

Definition: ERFC finds the complement of the error function (ERF) for a given real value and returns the result to the point of invocation.

Reference: ERFC (x)

Argument: The argument, "x," represents the value whose error function complement is to be found; "x" must be real.

Result: The value returned by this function is defined as follows:

ERFC(x) = 1-ERF(x)

EXP Mathematical Built-in Function

Definition: EXP raises e (the base of the natural logarithm system) to a given power and returns the result to the point of invocation.

Reference: EXP (x)

Argument: The argument, "x," specifies the power to which e is to be raised.

Result: The value returned by this function is e raised to the power of "x."

LOG Mathematical Built-in Function

Definition: LOG finds the natural logarithm (i.e., base e) of a given value and returns it to the point of invocation.

Reference: LOG (x)

Argument: The argument, "x," is the value whose natural logarithm is to be found. If "x" is real, it must not be less than or equal to 0; if "x" is complex, it must not be equal to 0+0i.

Result: The value returned by this function is the natural logarithm of "x." However, if "x" is complex, the function is multiple-valued; hence, only the principal value can be returned. The principal value has the form w = u±i*v, where v lies in the range:

$$-pi<v\leq pi$$

LOG10 Mathematical Built-in Function

Definition: LOG10 finds the common logarithm (i.e., base 10) of a given real value and returns it to the point of invocation.

Reference: LOG10 (x)

Argument: The argument, "x," represents the value whose common logarithm is to be found; this value must be real and it must not be less than or equal to 0.

Result: The value returned by this function is the common logarithm of "x."

LOG2 Mathematical Built-in Function

Definition: LOG2 finds the binary (i.e., base 2) logarithm of a given real value and returns it to the point of invocation.

Reference: LOG2 (x)

Argument: The argument, "x," is the value whose binary logarithm is to be found; it must be real and it must not be less than or equal to 0.

Result: The value returned to this function is the binary logarithm of "x."

SIN Mathematical Built-in Function

Definition: SIN finds the sine of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: SIN (x)

Argument: The argument, "x," is the value whose sine is to be found; it must be expressed in radians.

Result: The value returned by this function is the sine of "x." For complex argument "x," the sine of "x" is defined below, where $x = y_1+i*y_2$:

$$\sin(x)=\sin(y_1)*\cosh(y_2)+i*\cos(y_1)*\sinh(y_2)$$

SIND Mathematical Built-in Function

Definition: SIND finds the sine of a given real value, which is expressed in degrees, and returns the result to the point of invocation.

Reference: SIND (x)

Argument: The argument, "x," is the value whose sine is to be found; "x" must be real and it must be expressed in degrees.

Result: The value returned by this function is the sine of "x."

SINH Mathematical Built-in Function

Definition: SINH finds the hyperbolic sine of a given value and returns the result to the point of invocation.

Reference: SINH (x)

Argument: The argument, "x," is the value whose hyperbolic sine is to be found.

Result: The value returned by this function is the hyperbolic sine of "x." For complex argument "x," the hyperbolic sine of "x" is defined below, where $x = y_1+i*y_2$:

$\sinh(x)=\sinh(y_1)*\cos(y_2)+i*\cosh(y_1)*\sin(y_2)$

SQRT Mathematical Built-in Function

Definition: SQRT finds the square root of a given value and returns it to the point of invocation.

Reference: SQRT (x)

Argument: The argument, "x," is the value whose square root is to be found. If "x" is real, it must not be less than 0.

Result: If "x" is real, the value returned by this function is the positive square root of "x." If "x" is complex, the square root function is multiple-valued; hence, only the principal value can be returned to the user. The principal value has the form $w = u\pm i*v$, where either $u>0$, or $u=0$ and $v\geq 0$.

TAN Mathematical Built-in Function

Definition: TAN finds the tangent of a given value, which is expressed in radians, and returns it to the point of invocation.

Reference: TAN (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in radians.

Result: The value returned by this function is the tangent of "x."

TAND Mathematical Built-in Functions

Definition: TAND finds the tangent of a given real value which is expressed in degrees, and returns the result to the point of invocation.

Reference: TAND (x)

Argument: The argument, "x," represents the value whose tangent is to be found; "x" must be expressed in degrees.

Result: The value returned by this function is the tangent of "x."

TANH Mathematical Built-in Function

Definition: TANH finds the hyperbolic tangent of a given value and returns the result to the point of invocation.

Reference: TANH (x)

Argument: The argument, "x," represents the value whose hyperbolic tangent is to be found.

Result: The value returned by this function is the hyperbolic tangent of "x."

Summary of Mathematical Functions

Table 3 summarizes the mathematical built-in functions. In using it, the reader should be aware of the following:

1. A complex argument, "x," is defined as $x = y_1+i*y_2$.
2. The value returned by each function is always in floating-point.
3. The error conditions are those defined by the PL/I Language.
4. All arguments must be coded arithmetic and floating-point scale, or such that they can be converted to coded arithmetic and floating-point.

Table 3. Mathematical Built-in Functions

Function Reference	Argument Type	Value Returned	Error Conditions
ATAN(x)	real	arctan(x) in radians -(pi/2)<ATAN(x)<pi/2	-
	complex	-i*ATANH(i*x)	x = ±1i
ATAN(x,y)	both real	see function description	error if x=0 and y=0
ATAND(x)	real	arctan(x) in degrees -90<ATAND(x)<90	-
ATAND(x,y)	both real	see function description	error if x=0 and y=0
ATANH(x)	real	arctanh(x)	ABS(x)≥1
	complex	(LOG((1+x)/(1-x)))/2	x = ±1
COS(x) <u>x</u> in radians	real	cosine(x)	-
	complex	cos(y_1)*cosh(y_2) -i*sin(y_1)*sinh(y_2)	-
COSD(x) <u>x</u> in degrees	real	cosine(x)	-
COSH(x)	real	cosh(x)	-
	complex	cosh(y_1)*cos(y_2) +i*sinh(y_1)*sin(y_2)	-
ERF(x)	real	$\frac{2}{\sqrt{\Pi}}\int_0^x e^{-t^2}dt$	-
ERFC(x)	real	1 - ERF(x)	-
EXP(x)	real	e^x	-
	complex	e^x	-
LOG(x)	real	log (x)	x≤0
	complex	log (x) = w where w = u±i*v and -pi<v≤pi	x=0
LOG10(x)	real	$\log_1$ (x)	x≤0
LOG2(x)	real	$\log_2$(x)	x≤0

Table 3. Mathematical Built-in Functions (continued)

Function Reference	Argument Type	Value Returned	Error Conditions
SIN(x) x in radians	real	sin(x)	-
	complex	$\sin(y_1)*\cosh(y_2)$ $+i*\cos(y_1)*\sinh(y_2)$	-
SIND(x) x in degrees	real	sin(x)	-
SINH(x)	real	sinh(x)	-
	complex	$\sinh(y_1)*\cos(y_2)$ $+i*\cosh(y_1)*\sin(y_2)$	-
SQRT(x)	real	$\sqrt{x}$	x<0
	complex	$x = \sqrt{w}$ where $w = u \pm i*v$ and either $u>0$, or $u=0$ and $v \geq 0$	-
TAN(x) x in radians	real	tangent(x)	-
	complex	tangent(x)	-
TAND(x) x in degrees	real	tangent(x)	-
TANH(x)	real	tanh(x)	-
	complex	tanh(x)	-

ARRAY MANIPULATION BUILT-IN FUNCTIONS

The built-in functions described here may be used for the manipulation of arrays. All of these functions require array arguments (which may be expressions) and return single element values. Note that since these functions return element values, a function reference to any of them is considered an element expression.

ALL Array Manipulation Function

Definition: ALL tests all bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not the corresponding bits of given array elements are all ones.

Reference: ALL (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in "x" and whose bit values are determined by the following rule:

> If the ith bits of all of the elements in "x" exist and are 1, then the ith bit of the result is 1; otherwise, the ith bit of the result is 0.

ANY Array Manipulation Function

Definition: ANY tests the bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not at least one of the corresponding bits of the given array elements is set to 1.

Reference: ANY (x)

Argument: The argument, "x," is an array of bit strings. If the elements are not bit strings, they are converted to bit strings.

Result: The value returned by this function is a bit string whose length is equal to the length of the longest element in "x" and whose bit values are determined by the following rule:

> If the *i*th bit of any element in "x" exists and is 1, then the *i*th bit of the result is 1; otherwise, the *i*th bit of the result is 0.

DIM Array Manipulation Function

Definition: DIM finds the current extent for a specified dimension of a given array and returns it to the point of invocation.

Reference: DIM (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x," the extent of which is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision, giving the current extent of the *n*th dimension of "x."

HBOUND Array Manipulation Function

Definition: HBOUND finds the current upper bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: HBOUND (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x" for which the upper bound is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision giving the current upper bound for the *n*th dimension of "x."

LBOUND Array Manipulation Function

Definition: LBOUND finds the current lower bound for a specified dimension of a given array and returns it to the point of invocation.

Reference: LBOUND (x,n)

Arguments: The argument "x" is the array to be investigated; "n" is the dimension of "x" for which the lower bound is to be found. If "n" is not a binary integer, it is converted to a binary integer of default precision. It is an error if "x" has less than "n" dimensions, if "n" is less than or equal to 0, or if "x" is not currently allocated.

Result: The value returned by this function is a binary integer of default precision giving the current lower bound of the *n*th dimension of "x."

POLY Array Manipulation Function

Definition: POLY forms a polynomial from two given arguments and returns the result of the evaluation of that polynomial to the point of invocation.

Reference: POLY (a,x)

Arguments: Arguments "a" and "x" must be one-simensional arrays (vectors). They are defined as follows:

    a(m:n)

    x(p:q)

where (m:n) and (p:q) represent the bounds of "a" and "x," respectively.

Result: The value returned by this function is defined as:

$$a(m) + \sum_{j=1}^{n-m} (a(m+j) * \prod_{i=0}^{j-1} x(p+i))$$

If (q-p)<(n-m-1), then x(p+i)=x(q) for (p+i)>q. If m=n, then the result is a(m).

If "x" is an element variable, it is interpreted as an array of one element, i.e., x(1), and the result is then:

$$\sum_{j=0}^{n-m} a(m+j)*x**j$$

PROD Array Manipulation Function

Definition: PROD finds the product of all of the elements of a given array and returns that product to the point of invocation.

Reference: PROD (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being multiplied with the previous product.

Result: The value returned by this function is the product of all of the elements in "x." The scale of the result is floating-point, while the base, mode, and precision are those of the converted elements of "x."

SUM Array Manipulation Function

Definition: SUM finds the sum of all of the elements of a given array and returns that sum to the point of invocation.

Reference: SUM (x)

Argument: The argument, "x," should be an array of coded arithmetic floating-point elements. If it is not, each element is converted to coded arithmetic and floating-point before being summed with the previous total.

Result: The value returned by this function is the sum of all of the elements in "x." The scale of the result is floating-point, while the base, mode, and precision are those of the converted elements of the argument.

CONDITION BUILT-IN FUNCTIONS

The condition built-in functions allow the PL/I programmer to investigate interrupts that arise from enabled ON-conditions. None of these functions requires arguments. Each condition built-in function returns the value described only when executed in an on-unit (or a block activated directly or indirectly by an on-unit) that is entered as a result of an interrupt caused by one of the ON-conditions for which the function can be used. Such an on-unit can be one specific to the condition, or it can be for the ERROR or FINISH condition when these conditions are raised as standard system action. If a condition built-in function is used out of context, the value returned is as described for each function.

The on-units in which each function can be used are given in the function definition.

DATAFIELD Condition Built-in Function

Definition: Whenever the NAME condition is raised, DATAFIELD may be used to extract the contents of the data field that caused the condition to be raised. It can be used only in an on-unit for the NAME condition or in an ERROR or FINISH condition raised as a result of standard system action for the NAME condition.

Reference: DATAFIELD

Result: The value returned by this function is a varying-length character string giving the contents of the data field that caused the NAME condition to be raised. If DATAFIELD is used out of context, a null string is returned.

ONCHAR Condition Built-in Function

Definition: Whenever the CONVERSION condition is raised, ONCHAR may be used to extract the character the caused that condition to be raised. It can be used only in an on-unit for the CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for the CONVERSION condition. (ONCHAR can also be used as a pseudo-variable.)

Reference: ONCHAR

Result: The value returned by this function is a character string of length 1, containing the character that caused the CONVERSION condition to be raised. This character can be modified in the on-unit by the use of the ONCHAR or ONSOURCE pseudo-variables. If ONCHAR is used out of context, a blank is returned.

ONCODE Condition Built-in Function

Definition: ONCODE can be used in any on-unit to determine the type of interrupt that caused the on-unit to become active.

Reference: ONCODE

Result: ONCODE returns a binary integer of default precision. This "code" defines the type of interrupt that caused the entry into the currently active on-unit. If ONCODE is used out of context, an implementation-defined binary integer of default precision is returned.

ONCOUNT Condition Built-In Function

Definition: ONCOUNT can be used in any on-unit entered due to the abnormal completion of an input/output event to determine the number of interrupts (including the current one) that remain to be handled when a multiple interrupt has resulted from that abnormal completion.

Reference: ONCOUNT

Result: ONCOUNT returns a binary value of default precision. If ONCOUNT is used in an on-unit entered as part of a multiple interrupt, this value specifies the corresponding number of equivalent single interrupts (including the current one) that remain to be handled; if ONCOUNT is used in any other case, the returned value is zero.

ONFILE Condition Built-in Function

Definition: ONFILE determines the name of the file for which an input/output or CONVERSION condition was raised and returns that name to the point of invocation. This function can be used in the on-unit for any input/output or CONVERSION condition; it also can be used in an on-unit for an ERROR or FINISH condition raised as standard system action for an input/output or CONVERSION condition.

Reference: ONFILE

Result: The value returned by this function is a varying-length character string consisting of the name of the file for which an input/output or CONVERSION condition was raised. In the case of a CONVERSION condition, if that condition is not associated with a file, the returned value is the null string.

ONKEY Condition Built-in Function

Definition: ONKEY extracts the value of the key for the record that caused an input/output condition to be raised. It also extracts the key of a record in which a CONVERSION condition occurred during assignment specified by a KEYTO option. This function can be used in the on-unit for an input/output condition or a CONVERSION condition; it can also be used in an on-unit for an ERROR or FINISH condition raised as standard system action for one of the above conditions.

Reference: ONKEY

Result: The value returned by this function is a varying-length character string giving the value of the key for the record that caused an input/output or CONVERSION condition to be raised. If the interrupt is not associated with a keyed record, the returned value is the null string.

ONLOC Condition Built-in Function

Definition: Whenever an ON-condition is raised, ONLOC may be used in the on-unit for that condition to determine the entry point to the procedure in which that condition was raised. ONLOC may be used in any on-unit.

Reference: ONLOC

Result: The value returned by this function is a varying-length character string giving the name of the entry point to the procedure in which the ON-condition was raised. If ONLOC is used out of context, a null string is returned.

ONSOURCE Condition Built-in Function

Definition: Whenever the CONVERSION condition is raised, ONSOURCE may be used to extract the contents of the field that was being processed when the condition was raised. This function can be used in the on-unit for a CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition. (ONSOURCE can also be used as a pseudo-variable.)

Reference: ONSOURCE

Result: The value returned by this function is a varying-length character string giving the contents of the field being processed when CONVERSION was raised. This string may be modified in the on-unit by use of the ONCHAR or ONSOURCE pseudo-variable. If ONSOURCE is used out of context, a null string is returned.

BASED STORAGE BUILT-IN FUNCTIONS

The based storage built-in functions generally return special values to program control variables concerned in the use of based storage and list processing.

ADDR Based Storage Built-in Function

Definition: ADDR finds the location at which a given variable has been allocated and returns a pointer value to the point of invocation. The pointer value identifies the location at which the variable has been allocated.

Reference: ADDR (x)

Argument: The argument, "x," is the variable whose location is to be found. It can be any variable that represents an element, an array which is not interleaved, a structure, an area, an element of an array, a minor structure, or an element of a structure. It can be of any data type and storage class.

Result: ADDR returns a pointer value identifying the location at which "x" has been allocated. If "x" is a parameter, the returned value identifies the corresponding argument (dummy or otherwise). If "x" is an unallocated controlled variable, a null pointer value is returned.

EMPTY Based Storage Built-in Function

Definition: EMPTY clears an area of storage defined by an area variable, by effectively freeing all the allocations contained within the area. The area can then be used for a new set of allocations.

Reference: EMPTY

Arguments: None

Result: EMPTY returns an area of zero size, containing no allocations, to the point of invocation. When this value is assigned to an area variable, all the allocations contained within the area are freed.

Note: The value of the EMPTY built-in function is automatically assigned to all area variables when they are allocated.

NULL Based Storage Built-in Function

Definition: NULL returns a null pointer value (that is, a pointer value that cannot identify any allocation) so as to indicate that a pointer variable does not currently identify an allocation.

Reference: NULL

Arguments: None

Result: The value returned by this function is a null pointer value. This value cannot be converted to offset type.

NULLO Based Storage Built-in Function

Definition: NULLO returns a null offset value (that is, an offset value that cannot identify any relative location of a based variable allocation) so as to indicate that an offset variable does not currently identify an allocation.

Reference: NULLO

Arguments: None

Result: The value returned by this function is a null offset value. This value cannot be converted to pointer type.

OFFSET Based Storage Built-in Function

Definition: OFFSET returns an offset value relative to the beginning of a specified area.

Reference: OFFSET (p,a)

Arguments: The argument, "p," is a scalar pointer expression; "a" is a scalar area expression that may be qualified and/or subscripted. The value of "p" must identify an allocation in "a."

Result: The value returned by the OFFSET built-in function is an offset value that identifies an allocation in "a," relative to the beginning of "a."

POINTER Based Storage Built-in Function

Definition: POINTER returns a pointer value that identifies an allocation in a specified area.

Reference: POINTER (o,a)

Arguments: The argument, "o," is an offset expression; "a" is a scalar area expression that may be qualified and/or subscripted. The value of "o" must identify an equivalent allocation in some area, but not necessarily in "a."

Result: The value returned by the POINTER built-in function is a pointer value that identifies, in "a," a generation equivalent to the allocation originally identified by the offset "o."

MULTITASKING BUILT-IN FUNCTIONS

The multitasking built-in functions are used during multitasking and during asynchronous input/output operations. They allow the programmer to investigate the relative priority of a task or the current state of execution of a task or asynchronous input/output operation. They all require arguments, which may be scalar variables or aggregates.

COMPLETION Multitasking Built-in Function

Definition: COMPLETION determines the completion value of a given event variable. (COMPLETION can also be used as a pseudo-variable.)

Reference: COMPLETION(event-name)

Argument: The argument, "event-name," represents the event (or events) whose completion value is to be determined. The event can be associated with completion of a task, or with completion of an input/output operation, or it can be user-defined. It can be active or inactive.

Result: The value returned by this function is '0'B if the event is incomplete, '1'B if the event is complete.

PRIORITY Multitasking Built-in Function

Definition: PRIORITY determines the relative priority of a given task. (PRIORITY can also be used as a pseudo-variable.)

Reference: PRIORITY (task-name)

Argument: The argument, "task-name," represents the task whose relative priority is to be determined.

Result: The value returned by this task is a fixed binary value of precision (n,0), where n is implementation-defined. The value is the priority value of the named task, relative to the priority of the task evaluating the function. No other task can interrupt and gain control during evaluation of the priority.

STATUS Multitasking Built-in Function

Definition: STATUS determines the status value of a given event variable. (STATUS can also be used as a pseudo-variable.)

Reference: STATUS (event-name)

Argument: The argument, "event-name", represents the event (or events) whose status value is to be determined. The event can be associated with completion of a task, or with completion of an input/output operation, or it can be user-defined. It can be active or inactive.

Result: The value returned by this function is a fixed binary value of default precision. It is zero if the event is normal, or nonzero if abnormal.

MISCELLANEOUS BUILT-IN FUNCTIONS

The functions described in this section have little in common with each other and with the other categories of built-in functions. Some require arguments and others do not.

ALLOCATION Built-in Function

Definition: ALLOCATION determines whether or not storage has been allocated for a given controlled variable and returns an appropriate indication to the point of invocation.

Reference: ALLOCATION (x)

Argument: The argument, "x," must be a level 1 unsubscripted controlled variable.

Result: The value returned by this function is defined as follows: if an allocation of "x" is known in the current task, the returned value is '1'B; if no allocation is known, the returned value is '0'B.

COUNT Built-in Function

Definition: COUNT determines the number of data items that were transmitted during the last GET or PUT operation on a given file and returns the result to the point of invocation.

Reference: COUNT (file-name)

Argument: The argument, "file name," represents the file to be investigated. This file must have the STREAM attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the number of element data items that were transmitted during the last GET or PUT operation on "file name." Note that if an on-unit or procedure is entered during a GET or PUT operation, and within that on-unit or procedure a GET or PUT is executed for the same file, the value of COUNT is reset for the new operation and is not restored when the original GET or PUT is continued.

DATE Built-in Function

Definition: DATE returns the current date to the point of invocation.

Reference: DATE

Arguments: None

Result: The value returned by this function is a character string of length six, in the form yymmdd, where:

yy is the current year

mm is the current month

dd is the current day

LINENO Built-in Function

Definition: LINENO finds the current line number for a file having the PRINT attribute and returns that number to the point of invocation.

Reference: LINENO (file-name)

Argument: The argument, "file name," must be the name of a file having the PRINT attribute.

Result: The value returned by this function is a binary fixed-point integer of default precision specifying the current line number of "file name."

TIME Built-in Function

Definition: TIME returns the current time to the point of invocation.

Reference: TIME

Arguments: None

Result: The value returned by this function is a character string of length nine, in the form hhmmssttt, where:

hh is the current hour of the day

mm is the number of minutes

ss is the number of seconds

ttt is the number of milliseconds

PSEUDO-VARIABLES

In general, pseudo-variables are certain built-in functions that can appear wherever other variables can appear in order to receive values. In short, they are built-in functions used as receiving fields. A pseudo-variable may appear on the left of the equal sign in an assignment or DO statement; it may appear in the data list of a GET statement; it may appear as the string name in the KEYTO, STRING and REPLY options.

Since all pseudo-variables have built-in function counterparts, only a short description of each pseudo-variable is given here; the discussion of the corresponding built-in function should be consulted for the details. Note that pseudo-variables cannot be nested; for example, the following statement is invalid:

```
UNSPEC(SUBSTR(A,1,2))='00'B;
```

COMPLETION Pseudo-Variable

Reference: COMPLETION (event-name)

Description: The named scalar or aggregate event variable must be inactive and is as described for the COMPLETION built-in function. The value received by this pseudo-variable is a bit-string of length 1. This value sets the completion status of the "event variable." A value of '0'B specifies that the event associated with the "event variable" is incomplete; a value of '1'B specifies that the event is complete. Assignment to the pseudo-variable is uninterruptible.

COMPLEX Pseudo-Variable

Reference: COMPLEX (a,b)

Description: Only complex values can be assigned to this pseudo-variable. The real part of the complex value is assigned to the variable "a"; the imaginary part is assigned to the variable "b." The attributes of "a" and "b" need not be the same. Either or both arguments may be aggregates.

IMAG Pseudo-Variable

Reference: IMAG (c)

Description: Real or complex values may be assigned to this pseudo-variable. The real value or the real part of the complex value is assigned to the imaginary part of the complex variable "c," which may be an element variable or an array variable.

ONCHAR Pseudo-Variable

Reference: ONCHAR

Description: ONCHAR can be used in the on-unit for a CONVERSION condition or in the on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONCHAR is used in some other context, it is an error.

The expression being assigned to ONCHAR is evaluated, converted to a character string of length 1, and assigned to the character that caused the error. The new character will displace the current value of the ONCHAR built-in function, and will be used when the conversion is re-attempted, upon the resumption of execution at the point of interrupt.

ONSOURCE Pseudo-Variable

Reference: ONSOURCE

Description: ONSOURCE can be used in the on-unit for a CONVERSION condition or in an on-unit for an ERROR or FINISH condition raised as standard system action for a CONVERSION condition; it can also be used in a block directly or indirectly activated by such an on-unit. If ONSOURCE is used in some other context, it is an error.

The expression being assigned to ONSOURCE is evaluated, converted to a character string, and assigned to the string that caused the CONVERSION condition to be raised. The string will be padded with blanks, if necessary, to match the length of the string that caused the error. This new string displaces the current value of the ONSOURCE built-in function and will be used when the conversion is re-attempted, upon the resumption of execution at the point of interrupt.

PRIORITY Pseudo-Variable

Reference: PRIORITY [(task-name)]

Description: The "task-name" is as described for the PRIORITY built-in function, but need not be specified. The value received by this pseudo-variable is a fixed-point binary value _m_ of precision (n,0), where _n_ is implementation-defined. The priority value of the named task variable is adjusted so that it becomes _m_ relative to the priority that the current task had prior to the assignment. If an active task is associated with the named task variable, its priority is given the same value as the task variable.

If "task-name" is not specified, the task variable associated with the current task (if there is such a variable) is implied, and the priority of this variable is modified; hence, the priority of the current task is modified.

Assignment to the PRIORITY pseudo-variable is uninterruptible.

REAL Pseudo-Variable

Reference: REAL (c)

Description: Real or complex values may be assigned to this pseudo-variable. The real value or the real part of the complex value is assigned to the real part of the complex variable "c," which may be an element variable or an aggregate variable.

STATUS Pseudo-Variable

Reference: STATUS (event-name)

Description: The named event variable or aggregate of event variables can be active or inactive, and is as described for the STATUS built-in function. The value received by this pseudo-variable is a fixed point binary value of default precision. Assignment to the pseudo-variable is uninterruptible.

SUBSTR Pseudo-Variable

Reference: SUBSTR (string,i[,j])

Description: The value being assigned to SUBSTR is assigned to the substring of the character- or bit-string variable "string," as defined for the built-in function SUBSTR. If "string" is an aggregate, i and/or j may be aggregates. The remainder of "string" remains unchanged.

UNSPEC Pseudo-Variable

Reference: UNSPEC (v)

Description: The letter "v" represents an element or aggregate variable of arithmetic, string, locator, or area type. The value being assigned to UNSPEC is evaluated, converted to a bit string (the length of which is a function of the characteristics of "v" -- see the UNSPEC built-in function), and then assigned to "v," without conversion to the type of "v." If "v" is a string of varying length, its length after the assignment will be the same as that of the bit string assigned to it.

Picture specification characters appear in either the PICTURE attribute or the P format item for edit-directed input and output. In either case, an individual character has the same meaning. The PICTURE attribute is described in Chapter 4 and the P format item is described in Chapter 7 of this publication.

Picture characters are used to describe the attributes of the associated data item, whether it is the value of a variable or a data item to be transmitted between the program and external storage.

A picture specification always describes a character representation that is either a character-string data item or a numeric character data item or a bit representation that is a numeric bit data item. A pictured character-string item is one that can consist of alphabetic characters, decimal digits, and other special characters. A pictured numeric character item is one in which the data itself can consist only of decimal digits, a decimal point and, optionally, a plus or minus sign. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are considered to be part of the character-string value of the variable. A pictured numeric bit item is one in which the data itself can consist only of binary digits, either signed or in 1's or 2's complement form, with an assumed binary point.

Arithmetic data assigned to a numeric character variable is converted to character representation. Editing, such as zero suppression and the insertion of other characters, can be specified for a numeric character data item. Editing cannot be specified for pictured character-string data.

Data assigned to a variable declared with a numeric picture specification, either decimal or binary, (or data to be written with a numeric picture format item) must be either internal coded arithmetic data or data that can be converted to coded arithmetic. Thus, assigned data can contain only digits and, optionally, a decimal point and a sign. It should not contain any other character, even though that character (for example, a currency symbol) is specified in the picture specification and is to be inserted into the data as part of its character-string value; if it does, the CONVERSION condition is raised.

Data assigned to a variable declared with a character-string picture specification (or data to be written with a character-string picture format item) should conform, character by character (or be convertible, character by character) to the picture specification; if it does not, the CONVERSION condition is raised.

PICTURE CHARACTERS FOR CHARACTER-STRING DATA

Only three picture characters can be used in character-string picture specifications:

X specifies that the associated position can contain any character whose internal bit configuration can be recognized by the computer in use.

A specifies that the associated position can contain any alphabetic character or a blank character.

9 specifies that the associated position can contain any decimal digit or a blank character.

No insertion characters can be specified. At least one A or X must appear.

PICTURE CHARACTERS FOR NUMERIC CHARACTER DATA

Numeric character data must represent numeric values; therefore, the associated picture specification cannot contain the characters X or A. The picture characters for numeric character data can specify detailed editing of the data.

A numeric character variable can be considered to have two different kinds of value, depending upon its use. They are (1) its arithmetic value and (2) its character-string value.

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, and possibly a sign. The arithmetic value of a numeric character variable is used whenever the variable appears in an expression that results in a coded arithmetic value or whenever the variable is assigned to a coded arithmetic, numeric character, or bit-string variable. In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

The character-string value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character-string value does not, however, include the assumed location of a decimal point, as specified by the picture character V. The character-string value of a numeric character variable is used whenever the variable appears in a character-string expression operation or in an assignment to a character-string variable, whenever the data is printed using list-directed or data-directed output, or whenever a reference is made to a character-string variable that is defined on the numeric character variable. In such cases, no data conversion is necessary.

The picture characters for numeric character specifications may be grouped into the following categories:

- Digit and Point Specifiers
- Zero Suppression Characters
- Insertion Characters
- Signs and Currency Symbol
- Credit, Debit, and Overpunched Signs
- Exponent Specifiers
- Scaling Factor
- Sterling Picture Characters

The picture characters in these groups may be used in various combinations. Consequently, a numeric character specification can consist of two or more parts such as a sign specification, an integer subfield, a fractional subfield and, for floating-point, an exponent field. A sterling picture specification contains separate fields for pounds, shillings, and pence; the pence field can have an integer subfield and a fractional subfield.

A major requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This picture character, however, need not be the digit character 9. Other picture characters, such as the zero suppression characters (Z or * or Y), also specify digit positions. At least one of these characters must be used to define a numeric character specification.

DECIMAL DIGIT AND POINT SPECIFIERS

The picture characters 9 and V are used in the simplest form of decimal numeric character specifications that represent fixed-point decimal values.

9 specifies that the associated position in the data item is to contain a decimal digit.

V specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point is to be inserted. The integer and fractional parts of the assigned value are aligned on the V character; therefore, an assigned value may be truncated or extended with zero digits at either end. (Note that if significant digits are truncated on the left, the result is undefined and a SIZE interrupt will occur, if SIZE is enabled.) If no V character appears in the picture specification of a fixed-point decimal value (or in the first field of a picture specification of a floating-point decimal value), a V is assumed at the right end of the field specification. This can cause the assigned value to be truncated, if necessary, to an integer. The V character cannot appear more than once in a picture specification. The V is considered to be a subfield delimiter in the picture specification; that is, the portion preceding the V and the portion following it (if any) are each a subfield of the specification.

BINARY DIGIT AND POINT SPECIFIERS

The picture characters 1, 2, 3, and V are used in numeric bit specifications to represent binary digits and a binary point.

1 specifies that the associated position in the data item is to contain a binary digit.

2 specifies that the associated position in the 2's complement data item is to contain a binary digit.

3 specifies that the associated position in the 1's complement data item is to contain a binary digit.

V specifies that a binary point is assumed at this position in the associated data item. Its effect is the same as that described above for the V picture character, as used in a numeric character picture specification.

ZERO SUPPRESSION CHARACTERS

The zero suppression picture characters specify conditional digit positions in the character-string value and may cause leading zeros to be replaced by asterisks or blanks and nonleading zeros to be replaced by blanks. Leading zeros are those that occur in the leftmost digit positions of fixed-point numbers or in the leftmost digit positions of the two parts of floating-point numbers, that are to the left of the assumed position of a decimal point, and that are not preceded by any of the digits 1 through 9. The leftmost nonzero digit in a number and all digits, zeros or not, to the right of it represent significant digits. Note that a floating-point number can also have a leading zero in the exponent field.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character. When the associated data position does not contain a leading zero, the digit in the position is not replaced by a blank character. The picture character Z cannot appear in the same subfield as the picture character *, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

* specifies a conditional digit position and is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character * cannot appear with the picture character Z in the same subfield, nor can it appear to the right of a drifting picture character or any of the picture characters 9, T, I, or R in a field.

Y specifies a conditional digit position and causes a zero digit, leading or nonleading, in the associated position to be replaced by a blank character. When the associated position does not contain a zero digit, the digit in the position is not replaced by a blank character.

Note: If one of the picture characters Z or * appears to the right of the picture character V, then all fractional digit positions in the specification, as well as all integer digit positions, must employ the Z or * picture character, respectively. When all digit positions to the right of the picture character V contain zero suppression picture characters, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The entire character-string value of the data item will then consist of blanks or asterisks. No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

INSERTION CHARACTERS

The picture characters comma (,), point (.), slash (/), and blank (B) are insertion characters; they cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit positions, but are inserted between digits. Each does, however, actually represent a character position in the character-string value, whether or not the character is suppressed. The comma, point, and slash are conditional insertion characters; within a string of zero suppression characters, they, too, may be suppressed. The blank (B) is an unconditional insertion character; it always specifies that a blank is to appear in the associated position.

Note: Insertion characters are applicable only to the character-string value. They specify nothing about the arithmetic value of the data item.

causes a comma to be inserted into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the comma is inserted only when an unsuppressed digit appears to the left of the comma position, or when a V appears immediately to the left of it and the fractional part contains any significant digits. In all other cases where zero suppression occurs, one of three possible characters is inserted in place of the comma. The choice of character to replace the comma depends upon the first picture character that both precedes the comma position and specifies a digit position:

- If this character position is an asterisk, the comma position is assigned an asterisk.

- If this character position is a drifting sign or a drifting currency symbol (discussed later), the drifting string is assumed to include the comma position, which is assigned the drifting character.

- If this character position is not an asterisk or a drifting character, the comma position is assigned a blank character.

is used the same way the comma picture character is used, except that a point (.) is assigned to the associated position. This character never causes point alignment in the picture specifications of a fixed-point decimal number and is not a part of the arithmetic value of the data item. That function is served solely by the picture character V. Unless the V actually appears, it is assumed to be to the right of the rightmost digit position in the field, and point alignment is handled accordingly, even if the point insertion character appears elsewhere. The point (or the comma or slash) can be used in conjunction with the V to cause insertion of the point (or comma or slash) in the position that delimits the end of the integer portion and the beginning of the fractional portion of a fixed-point (or floating-point) number, as might be desired in printing, since the V does not cause printing of a point. The point must immediately precede or immediately follow the V. If the point precedes the V, it will be inserted only if a significant digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it will be suppressed if all digits to the right of the V are suppressed, but it will appear if there are any fractional digits (along with any intervening zeros).

/ is used the same way the comma picture character is used, except that a slash (/) is inserted in the associated position.

B specifies that a blank character always be inserted into the associated position of the character-string value of the numeric character data.

SIGNS AND CURRENCY SYMBOL

The picture characters S, +, and - specify signs in numeric character data. The picture character $ specifies a currency symbol in the character-string value of numeric character data.

These picture characters may be used in either a static or a drifting manner. A drifting character is similar to a zero suppression character in that it can cause zero suppression. However, the character specified by the drifting string is always inserted in the position specified by the end of the drifting string or in the position immediately to the left of the first significant digit.

The static use of these characters specifies that a sign, a currency symbol, or a blank always appears in the associated position. The drifting use specifies that leading zeros are to be suppressed. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a currency symbol.

A drifting character is specified by multiple use of that character in a picture field. Thus, if a field contains one currency symbol ($), it is interpreted as static; if it contains more than one, it is interpreted as drifting. The drifting character must be specified in each digit position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, slash, or B. Any of the insertion characters slash, comma, point, or B following the last drifting symbol of the string is considered part of the drifting string. However, a following V terminates the drifting string and is not part of it. A field of a picture specification can contain only one drifting string. A drifting string cannot be preceded by a digit position. The picture characters * and Z cannot appear to the right of a drifting string in a field.

The position in the data associated with the characters slash, comma, point, and B appearing in a string of drifting characters will contain one of the following:

- slash, comma, point, or blank if a significant digit has appeared to the left

- the drifting symbol, if the next position to the right contains the leftmost significant digit of the field

- blank, if the leftmost significant digit of the field is more than one position to the right

If a drifting string contains the drifting character n times, then the string is associated with n-1 conditional digit positions. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. If a drifting string is specified for a field, the other potentially drifting characters can appear only once in the field, i.e., the other character represents a static sign or currency symbol.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield.

Only one type of sign character can appear in each field. An S, +, or - used as a static character can appear to the left of all digits in the mantissa and exponent fields of a floating-point specification and either to the right or left of all digit positions of a fixed-point specification.

In the case in which all digit positions after the V contain drifting characters, suppression in the subfield will occur only if all of the integer and fractional digits are zero. The resulting edited data item will then be all blanks, except for the rightmost digit position, which will contain the drifting character. If there are any significant fractional digits, the entire fractional portion will appear unsuppressed.

$ specifies the currency symbol. If this character appears more than once, it is a drifting character; otherwise it is a static character. The static character specifies that the character is to be placed in the associated position. The static character must appear either to the left of all digit positions in a field of a specification or to the right of all digit positions in a specification. See details above for the drifting use of the character.

S specifies the plus sign character (+) if the data value is ≥ 0, otherwise it specifies the minus sign character (-). The character may be drifting or static. The rules are identical to those for the currency symbol. The picture S is the only sign symbol that can be used in a binary picture specification, and it can be used only with the character 1.

+ specifies the plus sign character (+) if the data value is ≥ 0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

- specifies the minus sign character (-) if the data value is <0, otherwise it specifies a blank. The character may be drifting or static. The rules are identical to those for the currency symbol.

CREDIT, DEBIT, AND OVERPUNCHED SIGNS

The character pairs CR (credit) and DB (debit) specify the signs of real numeric character data items and usually appear in business report forms.

Any of the picture characters T, I, or R specifies an overpunched sign in the associated digit position of numeric character data. It indicates the sign of the arithmetic data item. Only one overpunched sign can appear in a specification for a fixed-point number. A floating-point specification can contain two, one in the mantissa field and one in the exponent field. The overpunch character can, however, be specified for any digit position within a field. The overpunched number then will appear in the specified digit position.

Note: When an overpunch character occurs in a P format item for edit-directed input, the corresponding character in the input stream may contain an overpunched sign.

CR specifies that the associated positions will contain the letters CR if the value of the data is less than zero. Otherwise, the positions will contain two blanks. The characters CR can appear only to the right of all digit positions of a field.

DB is used the same way that CR is used except that the letters DB appear in the associated positions.

T specifies that the associated position, on input, will contain a digit overpunched with the sign of the data. It also specifies that an overpunch is to be indicated in the character-string value.

I specifies that the associated position, on input, will contain a digit overpunched if the value is ≥ 0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is ≥ 0.

R specifies that the associated position, on input, will contain a digit overpunched if the value is <0; otherwise, it will contain the digit with no overpunching. It also specifies that an overpunch is to be indicated in the character-string value if the data value is <0.

Note: The picture characters CR, DB, T, I, and R cannot be used with any other sign characters in the same field.

EXPONENT SPECIFIERS

The picture characters K and E delimit the exponent field of a numeric character specification that describes floating-point decimal numbers. The exponent field is always the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

K specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E specifies that the associated position contains the letter E, which indicates the start of the exponent field. It cannot appear in a binary picture specification.

The value of the exponent is adjusted in the character-string value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

SCALING FACTOR

The picture character F specifies a scaling factor for fixed-point decimal numbers. It appears at the right end of the picture specification and is used in the following format:

```
F ([+|-] decimal-integer-constant)
```

F specifies that the optionally signed decimal integer constant enclosed in parentheses is the scaling factor. The scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the scaling factor is positive) or to the left (if negative) of its assumed position in the character-string value.

STERLING PICTURES

The following picture characters are used in picture specifications for sterling data:

8 specifies the position of a shilling digit in BSI single-character representation.

7 specifies the position of a pence digit in BSI single-character representation.

6 specifies the position of a pence digit in IBM single-character representation.

P specifies that the associated position contains the pence character D.

G specifies the start of a sterling picture. It does not specify a character in the numeric character data item.

H specifies that the associated position contains the shilling character S.

M specifies the start of a field. It does not specify a character in the numeric character data item.

Sterling data items are considered to be real fixed-point decimal data. When involved in arithmetic operations, they are converted to a value representing fixed-point pence. Sterling pictures have the general form:

```
PICTURE

      'G [editing-character-1] ...

      M pounds-field

      M [separator-1] ...
            shillings-field

      M [separator-2] ...
            pence-field

        [editing-character-2] ...'
```

"Editing character 1" can be one or more of the following static picture characters:

```
$   +   -   S
```

The "pounds field" can contain the following picture characters:

```
Z Y * 9 T I R , $ + - S
```

The last four characters ($ + - S) must be drifting characters. The comma can be used as an insertion character.

"Separator 1" can be one or more of the following picture characters:

```
          /  .  B
```

The "shillings field" can be:

```
    {99 | YY | ZZ | Y9 | Z9 | YZ | 8}
```

One of the nines can be replaced by T, I, or R, if no other sign indicator appears in any of the fields of the specification.

"Separator 2" can be one or more of the picture characters:

```
          /  .  B  H
```

The "pence field" takes the form:

```
  {99|YY|ZZ|Y9|7|Z9|YZ|6}

     [[V|V.|.V] 9|Z|Y]...
```

One of the nines can be replaced by T, I, or R, if no other sign indicator appears in any of the fields of the specification.

"Editing character 2" can be one or more of the picture characters $, +, -, or S and one or more of B or P, or CR or DB. A sign character or CR or DB can appear only if no other sign indicator appears in any of the fields of the specification.

The pounds, shillings, and pence fields must each contain at least one digit position.

Zero suppression in sterling pictures is performed on the total data item, not separately on each of the pounds, shillings, and pence fields. The Z picture character is not allowed to the right of a 6, 7, 8, or 9 picture character in a sterling specification. In sterling pictures, the field separator characters slash (/), point (.), B, and H are never suppressed.

APPENDIX 3: ON-CONDITIONS

The ON-conditions are those conditions that may be specified in the ON statement. These conditions are also specified in SIGNAL and REVERT statements.

For each condition name, the description in this appendix includes the circumstances under which the condition occurs, the standard system action that would be taken in the absence of programmer-specified action, and, where applicable, the result. ("Standard system action" does not refer to any operating system but to standard action prescribed for the language.)

For the conditions OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, or FIXEDOVERFLOW, an interrupt action will always take place on occurrence of the condition unless the occurrence is in a calculation lying within the scope of a prefix specifying NOOVERFLOW, NOUNDERFLOW, NOZERODIVIDE, NOCONVERSION, or NOFIXEDOVERFLOW. For the conditions SIZE, STRINGRANGE, SUBSCRIPTRANGE, or CHECK (identifier list), an interrupt will not take place on occurrence of the condition unless the occurrence is in a calculation lying within the scope of a prefix specifying the condition. (See "Prefixes," in Chapter 1).

For any other condition, whose name may not be used in a prefix, an interrupt always will result from the occurrence of the condition.

Multiple Interrupts

A multiple interrupt can occur only for an input/output operation that has been associated with an event variable. It occurs during the execution of the WAIT statement naming that event variable, if the event completed abnormally (i.e., if one or more conditions occurred during the operation). Since conditions for an input/output event are raised at the execution of the WAIT for that event, the interrupts for these conditions also occur at this time. It is possible for more than one interrupt to occur for an input/output event. The aggregate of interrupts for an input/output event is called a multiple interrupt.

When an input/output event completes abnormally, the order in which the conditions are raised, and therefore, the order in which the interrupts for these conditions occur, is implementation-defined. If the on-unit for such a condition issues an abnormal return, then all unprocessed conditions (i.e., remaining interrupts of the multiple interrupt) are ignored; if an on-unit issues a normal return, the next condition is processed. If an on-unit has not been established for such a condition (or if the on-unit is SYSTEM), the next condition outstanding will be processed only if the standard system action is to comment and continue; if the standard system action is otherwise, all remaining interrupts in the multiple interrupt will be ignored.

CLASSIFICATION OF CONDITIONS

The ON-conditions are classified as follows: computational conditions, input/output conditions, program-checkout conditions, list processing conditions, programmer-named conditions, and system-action conditions.

The computational conditions are associated with data handling, expression evaluation, and computation.

The input/output conditions are associated with data transmission.

The program-checkout conditions facilitate debugging of programs.

The list processing conditions are associated with area usage.

The programmer-named conditions permit the programmer to use conditions of his own naming. These conditions are raised only by a SIGNAL statement.

The system-action conditions provide facilities to the programmer to extend the standard system action taken after the occurrence of a condition or at the completion of a program.

COMPUTATIONAL CONDITIONS

CONVERSION: This condition is raised whenever an illegal conversion is attempted on character string data, either internally or during input or output. The condition will be raised for such errors as characters other than 0 or 1 in conversion to bit string, characters not permitted in conversion to numeric field, or illegal characters in conversion to arithmetic. The conversion is carried out character by character, and the condition is raised for each illegal conversion. This condition may also be raised when the number of digits in a floating-point exponent exceeds the number allowed by an implementation. On return from the on-unit for this condition, the conversion will be reattempted.

Result: When CONVERSION occurs, results of the entire resultfield are undefined.

Standard System Action: Comment and raise the ERROR condition.

FIXEDOVERFLOW: This condition occurs during fixed-point arithmetic operations if the results of these operations exceed N, the maximum field width as defined by the implementation. See SIZE for a related condition that occurs on assignment.

Result: Result of the invalid fixed-point operation is undefined.

Standard System Action: Comment and raise the ERROR condition.

OVERFLOW: This condition occurs when the exponent of a floating-point number exceeds the permitted maximum, as defined by the implementation.

Result: The value of such an invalid floating-point number is undefined.

Standard System Action: Comment and raise the ERROR condition.

SIZE: This condition is raised by conversions between data types, or between differing bases, scales, or precisions. The condition arises when a value is assigned to a data item or during input/output, with a loss of high-order bits or digits.

The SIZE condition should be distinguished from FIXEDOVERFLOW that occurs during arithmetic calculations. A value too large for the field to which it is assigned will raise a SIZE condition on assignment, regardless of whether there was a FIXEDOVERFLOW in the calculation of the value. FIXEDOVERFLOW depends upon the size of fixed-point numbers allowed in the implementation. SIZE depends upon the declared size or implementation-restricted size of the item of data receiving a value.

Result: The contents of the receiving field are undefined.

Standard System Action: Comment and raise the ERROR condition.

UNDERFLOW: This condition occurs when the exponent of a floating-point number is smaller than the permitted minimum, as defined by the implementation.

The condition does not occur when equal numbers are subtracted (often call significance error).

Result: The value of the floating-point number is set to zero.

Standard System Action: Comment and continue execution.

ZERODIVIDE: This condition occurs on an attempt to divide by zero. The condition does not distinguish between fixed-point and floating-point division; either can cause it.

Result: The result of division by zero is undefined.

Standard System Action: Comment and raise the ERROR condition.

INPUT/OUTPUT CONDITIONS

The following conditions are always enabled and cannot appear in prefix lists. If the same file is known in a program by more than one name (for example, a file parameter and its associated file argument), these names constitute a set of equivalent filenames. A condition specified for one filename applies to all names of the set. An on-unit established using one filename of the set can be overridden by specifying an on-unit for another filename of the set.

ENDFILE (filename): This condition may be raised during any GET or READ operation, and is caused by an attempt to read past a file delimiter. It indicates that there is no more data on the file.

The end-of-file status remains until the file is closed. Subsequent GET or READ statements will immediately raise the condition. On return from the on-unit, processing will continue at the next statement. If this condition is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same task.

Standard System Action: Comment and raise the ERROR condition.

ENDPAGE(filename): This condition is raised by a PUT statement when an attempt is made to start a new line beyond the limit specified for the current page by the

PAGESIZE option in an OPEN statement. This attempt may be made during data transmission (with associated format items, if edit-directed transmission), by the LINE option, or by the SKIP option. It is raised only once per page.

If this condition is raised during data transmission, then, on return from the on-unit, the data is written on the current line, which may have been changed by the on-unit. If it is raised by a LINE or SKIP option, then, on return from the on-unit, the action specified by LINE or SKIP is ignored.

When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option. During the execution of the on-unit for this condition, or after return from the on-unit without a PAGE option or PAGE format item having been specified, the line number may increase indefinitely. However, execution of a LINE option or a LINE format item specifying a line number less than or equal to the current line number will cause a result equivalent to that caused by the execution of a PAGE option. In this case, ENDPAGE will not be raised; however, since the current line is now one, ENDPAGE can be raised again.

Standard System Action: Start a new page.

TRANSMIT (filename): This condition may be raised during any input/output operation, and is caused by a permanent transmission error on the specified file. In STREAM input, it is raised after assignment to each data item or record which is potentially of incorrect value because of the transmission error. On return from the on-unit, processing will continue as if no error has occurred.

If this condition is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same task.

Standard System Action: Comment and raise the ERROR condition.

UNDEFINEDFILE (filename): This condition is raised whenever an attempt to open a file is unsuccessful. If the attempt is made through an OPEN statement, attempts to open all other files in that statement will be made before this condition is raised. If this condition is raised for more than one file in the same OPEN statement, on-units will be executed according to the order of appearance (taken from left to right) of the filenames in that OPEN statement. On return from the final on-unit, processing will continue with the next statement.

If this condition is raised by an implicit file opening in an input/output statement without the EVENT option, then upon normal return from the on-unit, processing continues with the remainder of the interrupted input/output statement. If the file was not opened in the on-unit, then the statement cannot be continued and the ERROR condition is raised.

If this condition is raised by an implicit file opening in an input/output statement having an EVENT option, then the interrupt occurs before the event variable is initialized. In other words, the event variable retains its previous value and remains inactive. On normal return from the on-unit, the event variable is initialized, that is, it is made active and its completion value is set to '0'B (provided the file has been opened in the on-unit). Processing then continues with the remainder of the interrupted statement. However, if the file has not been opened in the on-unit, the event variable remains uninitialized, the statement cannot be continued, and the ERROR condition is raised.

Standard System Action: Comment and raise the ERROR condition.

NAME (filename): This condition may be raised on data-directed GET statements. It is caused by an unrecognizable identifier in the input or by an identifier not in the associated data list. The condition is raised at the time the error occurs. On return from the on-unit, the execution of the GET statement is resumed with the next data field in the stream.

By using the DATAFIELD built-in function in the ON unit, the programmer may access the data field which contained the incorrect name.

Standard System Action: Ignore the field and comment.

KEY (filename): This condition may be raised by any keyed record operation. It is raised in the following cases:

1. A READ for which the key is not found
2. A WRITE or LOCATE for which the key already exists
3. A REWRITE for which the key is not found
4. A DELETE for which the key is not found
5. Specification of the character string representing the key is in conflict with the format prescribed by the implementation

If this condition is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same task.

On return from the on-unit, no further action is attempted, and control passes to the next statement.

Standard System Action: Comment and raise the ERROR condition.

RECORD (filename): This condition may be raised by any READ or REWRITE operation. It is raised when the record contains more or less data than the specified variable (i.e., the size of the variable differs from the actual record size). It may be raised on a WRITE when the implementation cannot execute the statement.

If this condition is raised by an input/output statement using the EVENT option, the interrupt does not take place until the execution of a subsequent WAIT statement for that event in the same task.

The ONCODE built-in function returns an indication of whether the record variable was less than or greater than the record in size.

Before the on-unit is invoked, the following action takes place:

1. If the variable cannot contain the record, the excess data of the record is lost.

2. If the variable is greater than the record in size, the excess data in the variable is not transmitted on output and is unaltered on input.

Standard System Action: Comment and raise the ERROR condition.

PROGRAM CHECKOUT CONDITIONS

SUBSCRIPTRANGE: This condition occurs when a subscript is evaluated and found to lie outside its specified bounds.

The condition does not distinguish between values that are too large and values that are too small.

Note that if more than one subscript is associated with an identifier, e.g., A(I,J,K), the occurrence of a SUBSCRIPTRANGE condition is signalled after each subscript has been checked.

Result: Undefined.

Standard System Action: Comment and raise the ERROR condition.

STRINGRANGE: This condition may be raised by any reference to the SUBSTR built-in function or pseudo-variable if the length specified for the substring is less than zero or if the substring does not lie entirely within (or correspond to) the specified string.

The condition can be raised only once for each scalar SUBSTR reference.

Result: On normal return from an on-unit, execution continues with a revised SUBSTR reference whose value is defined as follows:

Let $\underline{k}$ be the length of the first argument (after execution of the on-unit); the other two arguments are represented by $\underline{i}$ and $\underline{j}$

If $\underline{i}$ is greater than $\underline{k}$, the value is the null string

If $\underline{i}$ is less than or equal to $\underline{k}$, the value is that substring beginning with the $\underline{m}$th character or bit of the first argument, and extending $\underline{n}$ characters or bits, where $\underline{m}$ and $\underline{n}$ are defined by:

```
m = MAX(i,1)

n = MAX(0,MIN(j+MIN(i,1)-1,
    k-m+1))
    if j is specified

n = k-m+1
    if j is not specified
```

The values of $\underline{i}$ and $\underline{j}$ are established before entry to the on-unit; they are not reevaluated on return from the on-unit.

Standard System Action: Revise the SUBSTR reference, as described under "Result," comment, and continue.

CHECK (identifier-list): A statement prefix specifying this condition may only be applied to PROCEDURE or BEGIN statements.

In the identifier list, each identifier is one of the following:

a statement label constant
an unsubscripted variable name representing a scalar, array, or structure
an entry label

Note: The identifier list cannot contain based variables, parameters, or data having the DEFINED attribute.

Each item in the list is, in effect, enabled independently. It follows, therefore, that each item in the list can also be disabled independently. In other words, a REVERT statement can be used to change the ON action for one or more items in the identifier list.

If a structure identifier or an array of structures identifier appears in the identifier list of a CHECK prefix, such a prefix is equivalent to a CHECK prefix whose list contains, in the order in which they were declared, the base elements of that structure or array of structures. For example, if P is defined by

```
    DECLARE 1P,2Q,2R,2S;
then
    CHECK (P)
is equivalent to
    CHECK (Q,R,S)
```

Statement Label Constant: For a statement-label constant, the condition is raised prior to the execution of the statement to which the label is prefixed. If the label is prefixed to a non-executable statement, no condition will be raised.

Variables: For identifiers representing variables, the condition is raised whenever the value of the variable, or any generation of any part of the variable, has a value assigned to it by any statement within the scope of the prefix.

The condition will be raised by the explicit reference to an identifier ID in the circumstances listed below, where ID is:

an identifier in the list
an identifier representing a structure or element contained by, or containing, an identifier in the list

The reference to ID may be subscripted or qualified.

The condition will be raised for ID only if:

1. ID appears on the left hand side of an assignment statement. (This applies to assignment BY NAME even if the identifier mentioned does not appear in the final expansion of the statement.)

2. ID is set as a result of its appearance as an argument of a pseudo-variable that is used in a context for which CHECK is raised.

3. ID appears as the control variable of a DO statement (or ID is set as a result of a pseudo-variable appearing as the control variable of a DO loop).

4. ID appears in the data list of an edit-directed or list-directed GET statement.

5. ID is assigned a value by data-directed input. If ID is a structure, CHECK will be raised each time an element of that structure is assigned a value.

6. ID is specified as the receiving field in a REPLY, STRING, SET, INTO, or KEYTO option. If the statement specifying this option also has an EVENT option, the CHECK condition is raised for it at the time the WAIT statement is executed. Note that this applies to implied SET options, as well.

7. ID is passed as an argument, with no dummy created, and control returns to the invoking block other than by a GO TO statement.

However, the condition is NOT raised under any of the following circumstances:

1. If the value of a variable defined upon ID or upon part of ID changes value in any of the ways described above.

2. If a parameter or based variable which represents ID changes value.

3. If ID is set by the INITIAL attribute.

Each condition is raised after the statement which caused it to be raised has been executed. (Note that an IF statement is considered terminated just prior to the execution of the THEN or ELSE clause, and an ON statement just prior to the ON-unit specification.) If the statement has a task option, the condition is raised when the attaching task regains control. If the statement is a DO statement, the condition is raised each time control proceeds sequentially to the statement following the DO statement. If the DO specifies iteration, the condition is raised once for every iteration.

If a statement causes a CHECK condition to be raised for several identifiers, then the conditions will be raised in the left-to-right order of appearance of the identifiers in the statement.

Entry Names: For an entry name, the condition is raised prior to each invocation of the entry name. The condition is raised only if the entry name used in the invocation has CHECK enabled for it.

Result: Continue. The statement is executed normally.

Standard System Action: If the identifier is a statement label constant or a label, task, event, pointer, offset, or area variable or an entry name, only the identifier is printed on a debugging file.

If the identifier represents data other than that mentioned above, the identifier and its new value are both printed on a debugging file in the format of data-directed output.

LIST PROCESSING CONDITIONS

The following condition is always enabled and may not appear in a condition prefix.

AREA: This condition is raised when an attempt is made to allocate storage within an area defined by an area variable, and sufficient storage does not remain within the area. It can be raised by an ALLOCATE or assignment statement.

Result: On normal return from the on-unit, the reference to the area is reevaluated. Allocation is then attempted in this area.

Standard System Action: The ERROR condition is raised.

PROGRAMMER-NAMED CONDITIONS

CONDITION (identifier): This condition is always enabled and may not appear in a condition prefix. The identifier is specified by the programmer, and is EXTERNAL. The condition is raised by the execution of a SIGNAL statement having the same identifier.

Standard System Action: Comment and continue.

SYSTEM ACTION CONDITIONS

The following conditions are always enabled and cannot appear in a condition prefix.

FINISH: This condition is raised by execution of a statement that would cause termination of the major task of a PL/I program, that is, by a STOP in any task, an EXIT in the major task, or a RETURN or END in the initial external procedure of the major task. The condition also is raised by a SIGNAL FINISH statement in any task. The interrupt occurs in the task in which the statement is executed, and any on-unit specified for the condition is executed as part of that task.

An abnormal return from a FINISH on-unit will avoid any subsequent task termination processes and permit the interrupted task to continue.

On normal return from a FINISH on-unit, execution of the interrupted statement is resumed.

Standard System Action: Execution of the interrupted statement is resumed.

Note: When the FINISH condition is raised by execution of a SIGNAL FINISH statement, normal return from the FINISH on-unit (or standard system action, if no on-unit is established) will cause execution to continue with the statement following the a SIGNAL FINISH statement.

ERROR: This condition is raised either by a SIGNAL ERROR or by some error situation in the execution of the program. The abilities of different implementations to detect execution-time errors will vary; therefore, some of the conditions under which ERROR will be raised are implementation defined. An abnormal return from an ERROR on-unit will permit the interrupted task to continue execution. The action for normal return, however, is implementation defined.

Standard System Action: This action is implementation defined.

APPENDIX 4: PERMISSIBLE KEYWORD ABBREVIATIONS

Abbreviations are provided for certain keywords. The abbreviations themselves are keywords and will be recognized as synonomous in every respect with the full keywords. The abbreviated keywords are shown to the right of the full keywords in the following list.

ABNORMAL	ABNL
ACTIVATE	ACT
AUTOMATIC	AUTO
BINARY	BIN
BUFFERED	BUF
CHARACTER	CHAR
COLUMN	COL
COMPLETION	CPLN
COMPLEX	CPLX
CONTROLLED	CTL
CONVERSION	CONV
DEACTIVATE	DEACT
DECIMAL	DEC
DECLARE	DCL
DEFINED	DEF
ENVIRONMENT	ENV
EXCLUSIVE	EXCL
EXTERNAL	EXT
FIXEDOVERFLOW	FOFL
INITIAL	INIT
INTERNAL	INT
IRREDUCIBLE	IRRED
OVERFLOW	OFL
PICTURE	PIC
POINTER	PTR
POSITION	POS
PRECISION	PREC
PROCEDURE	PROC
REDUCIBLE	RED
SEQUENTIAL	SEQL
STRINGRANGE	STRG
SUBSCRIPTRANGE	SUBRG
UNALIGNED	UNAL
UNBUFFERED	UNBUF
UNDERFLOW	UFL
UNDEFINEDFILE	UNDF
VARYING	VAR
ZERODIVIDE	ZDIV

The characters that make up the 48-character set are the same as those that make up the 60-character set except for certain restrictions.

The following characters are not included:

Percent	%
Colon	:
Not	¬
Or	|
And	&
Greater Than	>
Less Than	<
Break character	_
Semicolon	;
Number sign	#
Commercial At sign	@
Question mark	?

The following three characters are replaced as indicated:

60-Character Set	48-Character Set
:	..
;	,.
%	//

The two periods which replace the colon must be immediately preceded by a blank if the preceding character is a period. The two slashes that replace the percent symbol must be immediately preceded by a blank if the preceding character is an asterisk, or immediately followed by a blank if the following character is an asterisk. The sequence "comma period" represents a semicolon except when it occurs in a comment or character string, or when it is immediately followed by a digit.

The following character combinations, as used in the 60-character set, are replaced in the 48-character set by alphabetic equivalents as indicated:

60-Character Set	48-Character Set
>	GT
¬>	NG
>=	GE
¬=	NE
<=	LE
<	LT
¬<	NL
¬	NOT
|	OR
&	AND
||	CAT
->	PT

The above words are "reserved" in the 48-character set; that is, they must not be used as programmer-specified identifiers.

In each case, one or more blanks must immediately precede the alphabetic operator if the preceding character would otherwise be alphameric, and one or more blanks must immediately follow if the following character would otherwise be alphameric. Thus, to indicate the comparison of the variables A6 and BQ2Y for inequality, one would write A6 NE BQ2Y, but not A6NEBQ2Y, A6 NEBQ2Y, or A6NE BQ2Y. As the equal symbol is usable, however, the comparison of these two variables for equality may be written A6=BQ2Y.

The break character, commercial at-sign, and number sign are not used and consequently may not be employed in identifiers.

READER'S COMMENT FORM

IBM System/360 Y33-6003-0

PL/I Language Specifications

- How did you use this publication?

As a reference source	☐
As a classroom text	☐
As a self-study text	☐

- Based on your own experience, rate this publication . . .

As a reference source:	Very Good	Good	Fair	Poor	Very Poor
As a text:	Very Good	Good	Fair	Poor	Very Poor

- What is your occupation? ....

- We would appreciate your other comments; please give specific page and line references where appropriate. If you wish a reply, be sure to include your name and address.

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

YOUR COMMENTS PLEASE . . . .

This SRL bulletin is one of a series which serves as reference sources for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

fold fold

FIRST CLASS
PERMIT NO. 1359
WHITE PLAINS, N.Y.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY...

IBM CORPORATION

112 EAST POST ROAD,
WHITE PLAINS, N.Y. 10601.

Attention: Department 813

fold fold

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]

Y33-6003-0

Printed in U.S.A. Y33-6003-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]