File No. S360-29
Form C28-6571-3

OS

IBM Systems Reference Library

IBM System/360 Operating System

PL/I Language Specifications

This manual is a description of the full facilities of PL/I to be implemented under the System/360 Operating System. However, the reader should not assume that all facilities will be available at initial release. Manuals for specific System/360 implementations will be released later.

Another publication will be issued specifying a subset of the facilities of the language described in this manual. This subset is planned for implementation under the System/360 Disk and Tape Operating System.

PREFACE

This publication is a reference manual for the entire PL/I Language. All of the features to be implemented under the System/360 Operating System are described herein.

However, this manual does not approach PL/I from a tutorial point of view. There are other IBM publications that perform this function. These publications and their intended audience are as follows:

1. 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.

2. 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.

3. A Guide to PL/I for Commerical 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.

Introductory information about PL/I may also be found in the Student Text, An Introduction to PL/I, Form C20-1632.

A familiarity with the contents of A PL/I Primer is recommended for the users of this reference manual.

MAJOR REVISION (JULY, 1966)

This publication, Form C28-6571-3, obsoletes the previous edition, Form C28-6571-2. New textual information and significant changes are identified by vertical lines to the left of the added and changed text.

Among the additions and significant changes are (1) the complete respecification of compile-time facilities (Chapter 9), (2) the new attributes REDUCIBLE and IRREDUCIBLE, and (3) the new data type called cell (and its corresponding CELL attribute).

Because Chapter 9 has been completely rewritten, vertical bars have not been used to indicate changes; this chapter should be re-read in its entirety.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.

A form for readers' comments appears at the back of this publication. It may be mailed directly to IBM. Address any additional comments concerning this publication to the IBM Corporation, Programming Systems Publications, Department D39, 1271 Avenue of the Americas, New York, N. Y., 10020.

CONTENTS

GOALS OF THE LANGUAGE

Throughout the relatively brief history of electronic data processing, certain computers have been identified with a particular field of activity, either commercial or scientific.

Programmers have generally specialized in one field or the other. High-level languages, of course, have emphasized this divergence, going in one direction for commercial programming and in another direction for scientific programming.

Until recently, this difference presented few problems. Each language was adequate for its use; the commercial programmer dealt with relatively few computations performed upon great amounts of data; the scientific programmer performed complex calculations using small amounts of data.

Now, however, the situation is changing. Business and industry have discovered new uses for the computer, and the commercial programmer finds himself concerned with more involved computations in statistical forecasting and in linear programming for operations research.

In science and engineering, the programmer needs a language to simplify the preparation of reports, to sort and edit technical data; he finds more need for input and output operations. The engineer specifically wants the ability to handle data at the bit level for applications such as circuit analysis.

Today's new computing systems have been designed to cope with all of these computing problems. They handle commercial and scientific programs with equal ease, with new power and new speed; they provide facilities for such new techniques as shared data processing, asynchronous program execution, and real-time processing.

None of the traditional high-level languages, however, can be used with efficiency across the entire range of ability of these new computers.

That is the reason for PL/I, a multipurpose programming language for use not only by commercial and scientific programmers but by the real-time programmer and the systems programmer as well. It is a language designed for efficiency, a language that enables the programmer to use virtually all the power of his computer.

PL/I is organized so that any programmer, no matter how extensive his experience, can use it easily at his own level.

This manual, because it is a reference manual of the entire language, shows the range and power of PL/I, its ability to handle the most complex computing problems.

Actually, however, PL/I need be no more complex than the program for which it is used.

One of the primary aims in the design of the language was modularity, that is, providing different levels of the language for different applications and different degrees of complexity. A programmer using one level need not even know about the unused facilities.

Although PL/I is relatively machine independent, this modularity might be compared to a completely equipped data processing center. A novice programmer would use only a small part of the system; he can ignore the rest of the equipment. More complex programs, of course, would require more equipment. Some programs would use certain modules of equipment; other programs, other modules. Rarely, if ever, would a program require use of all the facilities.

In PL/I, every attribute -- or description -- of a variable, every option, and every specification has been given a "default" interpretation. Wherever the language provides for one or more alternatives, a "default" interpretation -- or assumption -- is made by the compiler if no choice is stated by the programmer. And in each case, the assumption that was chosen in the design of the language is the one most likely to be required by the programmer who need not know that alternatives exist.

The "modularity" and the "default" aspects are the bases upon which the simplicity of PL/I has been built. They are also part of its power.

BASIC CHARACTERISTICS OF PL/I

The overall aim in the design of the language was to give the programmer freedom in handling his computing system.

Freedom of expression: If a particular combination of symbols has a useful meaning, that meaning is allowed. Although actual statements in the language must be written using a specified character set, data may be composed of any character allowed by the configuration of the individual computer. PL/I is written in a free-field format; the programmer is free to design his own format for listings.

Full access to machine and operating system facilities: the PL/I programmer rarely, if ever, will need to resort to assembly language coding.

SALIENT FEATURES

Part of the language is, of course, based on earlier programming languages. Certain aspects are expansions of ideas used previously. Other portions are exclusively a part of PL/I. The following paragraphs briefly describe some of these features. All of them are discussed more fully within the text.

Block Structure

The statements of a PL/I program are organized into program sections called "blocks." A program may be made up of one block or many blocks. Blocks may be separate from one another, with no common statements, or they may be nested, one within another.

Blocks provide two important logical functions: (1) they define the scope of applicability of data variables and other kinds of names, so that the same name may be used for different purposes in different blocks without ambiguity, and (2) they allow storage for data variables to be assigned only during execution of the block and freed for other uses at the termination of the block.

Certain blocks, called "procedure" blocks, may be invoked (i.e., called into execution) remotely from different places in the program, and they provide means to handle arguments and to return values.

Description of Data

In the language, data is described as having certain characteristics called attributes. For example, numeric data would have a BINARY attribute or a DECIMAL attribute; string data would be either CHARACTER string or BIT string.

Storage Allocation

The computer storage for any data variable in a PL/I program may be assigned statically, for the entire execution of the program, or dynamically, during execution.

Two classes of dynamic storage are available to the PL/I programmer, automatic and controlled. When a variable has the controlled storage attribute, the programmer may allocate or free storage for that variable at any time he wishes. Storage for a variable having the automatic storage attribute is allocated upon entry to the block and freed upon exit.

Data Conversion

In keeping with the freedom of PL/I, mixed expressions are allowed. In the following example, F is declared to be a fixed-point number, G a floating-point number, and H a character string that is ten characters in length.

```
DECLARE  F FIXED, G FLOAT, H CHARACTER
(10);
      H = F + G;
```

In the evaluation of the second statement of the above example, F will be converted to a floating-point value, floating-point addition will be performed, and the result will be converted to a character string of ten characters and assigned as a value to H.

Data Organization

Data variables can be grouped into either arrays or structures. An array is composed of elements of the same characteristics. A structure is a collection of variables and arrays, not necessarily alike in characteristics. Structures may also contain other structures. Individual items of an array are referred to by subscripted names; individual items of a structure are referred to by names that may sometimes have to be qualified to avoid ambiguity.

In PL/I, arrays and structures are treated as variables in their own right. Either of them may be used as the operand of an expression. The expression is then an array expression or a structure expression, and it returns an array or structure result.

Input/Output

The modularity of PL/I is particularly apparent in the input/output facilities. With PL/I, a programmer may control input/output activity to whatever degree he requires. He may handle normal transmission and conversion simply, or he may use the full capability of the language for control of more complex problems of input and output.

Multi-Task Operations

In PL/I, a collection of procedures is called a program; the execution of a program (or many programs or a part of a program) to perform a particular job is called a task.

PL/I provides facilities for handling two or more tasks concurrently. This facility, of course, is extremely important in the use of any computer system with multiprocessing capabilities. It also is valuable for a single processor system with facilities for real-time operations.

During execution of a procedure, the executing task might specify that a subordinate task begin execution upon certain data (i.e., the executing task invokes another task); the new task, called an attached task, might also invoke another task. All tasks then proceed concurrently and, in effect, simultaneously.

The multi-task facilities of PL/I allow a subordinate task to communicate with its originating, or attaching, task through arguments, and through data allocated in the attaching task. The originating task also may, at any time, test to see if a subordinate task is completed and may, if necessary, delay its own execution to wait for the completion.

Compile-Time Facilities

Most programming languages are written on one level only, as statements to the computer to perform certain operations using the supplied data. PL/I not only directs the computer to operate upon the data, but with a macro facility, it directs the compiler to operate upon the program itself.

The programmer can include in his program information that will aid the compiler to produce more efficient code, documentation, and diagnostics.

List Processing

PL/I provides facilities for list processing. These facilities are unusually flexible in that the introduction of pointer and based variables enables the programmer to combine arrays, structures, and scalars into a single list.

A complete enumeration of PL/I list processing facilities may be found under the heading "List Processing" in the Index (also see "List Processing" in Chapter 10).

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 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:

identifier { FIXED / FLOAT }

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 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 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 {&|⊥} operand
```

This denotes that the variables "operand" are separated by either an "and" (&) or an "or" (|). The constant | is underlined in order to distinguish the "or" symbol in

the PL/I language from the "or" symbols in the syntax language.

9. min max. The combination of these two words with associated numeric values specifies the minimum and maximum number of times a syntactical unit may occur. When min is used without max, the implied max is infinity. When max is used without min, the implied min is zero.

Examples:

a. min 2 max 6 {digit|letter}

This denotes that either "digit" or "letter" intermixed in any combination must occur at least two times, but no more than six.

b. min 5 {digit|letter}

The variables "digit" or "letter" intermixed in any combination must occur at least five times, but there is no limit on the number of times over five that they may occur.

c. max 3 label

The variable "label" may not occur more than three times in succession. It may not be present at all, or it may occur one, two, or three times.

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 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 may be used to create 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 delimiters and fall into three classes:

operators
parentheses
separators and other delimiters

Operators

Operators used by the language are divided into four types:

arithmetic operators
comparison operators
bit-string operators
string operators

Arithmetic Operators

The arithmetic operators are:

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

Comparison Operators

The comparison operators are:

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

Bit-String Operators

The bit-string operators are:

¬	denoting not
&	denoting and
|	denoting or

String Operator

The string operator is:

||	denoting concatenation

Parentheses

Parentheses 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	%	precedes macro statement
pointer qualification symbol	->	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

In the execution of PL/I programs, comparisons of character data will observe the collating sequence resulting from the representations of involved characters in bytes of System/360 storage, in extended binary coded decimal interchange code (EBCDIC).

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.

Identifiers in the language are used for the following:

scalar variable names

array names

structure names

statement labels

entry names

file names

keywords

condition names

Examples:

VARA

BCD320

FILE42

XR20A

STARTA

RATE_OF_PAY

#32_45

$L32

X@_52

@531

AB12#

Length of Identifiers

Identifiers that a programmer constructs in writing a PL/I program must be composed of not more than 31 characters.

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

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, picture specifications, composite operators (e.g., ¬=), and the class of dummy variables iSUB (see "The DEFINED Attribute" in Chapter 4) may not contain blanks. Blanks are permitted within a character-string constant.

Identifiers, constants, or picture specifications may not be immediately adjacent. They must be separated by an operator, assignment symbol (i.e., =), parenthesis, colon, semicolon, comma, period, blank, or comment. Moreover, additional intervening blanks or comments are always permitted. Blanks are optional between keywords of a statement identifier (e.g., GO TO), and between a level number and its following identifier (see "Structures" in Chapter 2).

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   (DO is the keyword)
  10;

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 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
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, ENTRY, or any compile-time statement.

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, 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 internal to). 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 known, see "Basic Rule on Use of Names" in Chapter 4).

Example:

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

In the example on the left above, 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 example on the right is equivalent to the example on the left.

The statement X:END B is internal to block B.

PROGRAMS

A program is composed of one or more external procedures. Thus, by definition, a program is a set of procedure blocks, each of which is completely nested, and separate from the others.

CHAPTER 2: DATA ELEMENTS

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).

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.

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 below. 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, 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 declaration. (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.

Structures may contain structures. The outermost structure is the major structure, and contained structures are minor structures. A major structure must be at level one. Contained structures must always have a level number numerically greater than the structure in which they are contained. Identifiers preceded by level numbers but having no components are not considered to be structures. The level number may be followed by one or more blanks. (Additional information on structures can be found in the section "Structure Declarations and Attributes" in Chapter 4.)

Examples:

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

   takes the form:

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

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:

```
X [ Y
    Z (1) [ P (1,1,1)
            P (1,1,2)
            P (1,2,1)
            P (1,2,2)
            Q (1)
    Z (2) [ P (2,1,1)
            P (2,1,2)
            P (2,2,1)
            P (2,2,2)
            Q (2)
  R
```

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 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 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)
```

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 $k \leq n$ 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) is the vector:
   ```
   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 qualified name. 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.

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 subscripted qualified names. 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. Unless all of the subscripts are moved to the lowest or highest level, the qualified name is said to have interleaved subscripts.

Provided that sufficient structure names are used to make the name unique, as described for qualified names, and that the total number of subscripts is the same as the total dimensionality of the array, unsubscripted structure names may be omitted in references. Ambiguity of names, however, cannot be resolved by subscripting. 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.

A subscripted qualified name must have at least one subscript.

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.

Except for forms (7) and (10), all of the qualified names in the above example have interleaved subscripts.

Example 2:

If FIELD is the array of structures:

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

then FIELD(*).STATUS is the vector:

```
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 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.

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 (w,d) is specified by giving the total number of binary or decimal digits, w, to be maintained and a scale factor, d. The precision of floating-point data is specified by giving the effective number, w, 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 w). Note that w 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:

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 that is 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 a string of 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 a string of 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 an apparent precision (w,d) where w is the total number of digits in the constant and d 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 the 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 string.

In a string replication factor, blanks may optionally surround the decimal integer constant, or they may separate the right parentheses 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 latter is exactly equivalent to

```
'TOMTOMTOM'
```

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 string.

Examples:

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

The latter is exactly equivalent to

```
'1111111111'B
```

String Variables

String variables are names of string data items. These names have been given the characteristics of string data (see Chapter 4).

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 is a statement-label constant. EXIT is also a statement-label.

Statement-Label Variables

A statement-label variable 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 statement-label designator.

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 (see "Asynchronous Operations and Tasks" in Chapter 6, and "The EVENT Attribute" in Chapter 4). An event variable may be an element of an array or of a structure.

An event variable has an associated completion status. This status is denoted by '0'B for "not completed" and '1'B for "completed." If the event variable has been associated with a given task via the use of the EVENT option in a CALL statement (see Chapter 8), the completion status of the event variable will reflect the completion status of the task itself. The completion status of an event variable may also be set explicitly by the execution of an assignment statement using the EVENT pseudo-variable (see Chapter 8).

Pointer Data

A pointer variable is the name of a pointer (see "The CONTROLLED Attribute" and "The POINTER Attribute" in Chapter 4, and "The Pointer Qualifier" in this chapter). It is used only in connection with list processing and RECORD transmission. A pointer variable may be an element of a structure or of an array.

Pointer Qualification

Pointer qualification is used to identify a generation of a based variable. This generation may also be identified by the pointer variable declared with the based variable (see "The CONTROLLED Attribute" in Chapter 4 and "The ALLOCATE Statement" in Chapter 8).

Format:

```
{scalar-pointer-variable|ADDR(variable)}->
   [{scalar-pointer-variable|
       ADDR(variable)}->]...based-variable
```

Note: See the ADDR built-in function for a discussion of ADDR.

General rules:

1. Pointer qualification is used to replace the pointer which was declared with the based variable.

2. More than one pointer qualifier may be specified in a reference. In this case, they are read left to right and define a chain of pointers qualifying the reference.

Examples:

```
P -> VALUE
P -> G -> VALUE
```

Area Data

An area data item is the name of an area of storage. Such an area may be used for collecting and referring to based data items (see "The ALLOCATE Statement" in Chapter 8).

Cell Data

A cell is a unit of storage that may contain any number of alternative declarations. However, only one declaration can be active at any one time.

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.

See "The CELL Attribute" in Chapter 4 for rules regarding cell usage.

EXPRESSIONS

An expression is an algorithm used for computing a value. Expressions are of the three types: scalar, array, and structure, depending upon the types of the operands involved. The type of the result is also the same as that of the operands. An array (or structure) expression is simply an array (or structure) evaluated by expansion of the expression into a collection of scalar expressions (see "Array Expressions" and "Structure Expressions"). Syntactically, a scalar expression consists of a constant, a scalar variable, a function reference, a scalar expression enclosed in parentheses, a scalar expression preceded by a prefix operator, or two scalar expressions connected by an infix operator. 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 type of the value is the type of the expression. The type of the expression is dependent upon the class of operators -- arithmetic, comparison, bit string, and concatenation (see "Operators"). Statement label designators, area variables, task variables, and event variables are not allowed in scalar expressions except as function arguments. Only the comparison operators = and ¬= may appear with pointer 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, 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.

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 first operand of an infix operation is 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 length of the largest number in the implementation
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 $x_2\leq 0$, the ERROR condition is raised.

c. If $x_1\neq 0$ 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 ERROR 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.

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

Table 1. Arithmetic Base and Scale Conversion

After	Before Conversion: 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),CEIL(ABS(q)*3.32)*SIGN(q))		
Decimal Fixed	(CEIL(p/3.32)+1, CEIL(ABS(q)/3.32)*SIGN (q))	(p,q)		
Binary Float	(p)	(MIN(CEIL (p*3.32),N))	(p)	(MIN(CEIL (p*3.32),N))
Decimal Float	(CEIL(p/3.32))	(p)	(CEIL(p/3.32))	(p)

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.)

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, truncation on both left and right will occur, and the SIZE error condition will be raised (unless disabled) if significant order digits are lost.

Bit-String Operations

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 is of varying length or is a reference to the SUBSTR built-in function. Otherwise, the result is of fixed length.

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. Algebraic, 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 a collating sequence. If the operands are of different length, 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.

If the operands of a comparison are of different types, the operand of the lower type is converted to the operand of the higher type. The priority of types is (1) arithmetic (highest), (2) character string, (3) bit string.

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

Only the operations = and ¬= are defined when either operand is complex.

Only the operators = and ¬= may be used with pointer variables. In this case, each operand must be either a pointer variable or a function that defines a pointer 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.

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 is illegal if the character string contains characters other than 0 and 1. 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.

The arithmetic value of the constant is converted to the base, scale, mode, and precision that a REAL FIXED DECIMAL value of default 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 converted to fixed-point binary, precision (S,0), where S depends upon the implementation. The null string is converted to the value zero.

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. See Appendix 1 also.

Arithmetic to Bit String

The 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,max(r-s,0))
BINARY FLOAT (r)     p = min(N,r)
DECIMAL FIXED (r,s)  p = min(N,max(CEIL
                       ((r-s)*3.32),0))
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 implementation-defined.

ARRAY EXPRESSIONS

An array expression is an expression consisting of array operands in possible combination with scalars and/or structures. Note that if a structure appears in an array expression, the array operands must be arrays of structures.

An array expression returns an array result. That is, all operations performed on arrays are performed on an element-by-element basis. Therefore, all arrays referred to in an array expression must be of identical bounds.

Note: Array expressions are not always expressions of conventional matrix algebra.

The appearance of a function reference (other than a built-in function) will imply a scalar result. For example, if A is an array, CALC(A) may be a scalar function with an array argument.

The built-in functions listed under "Arithmetic Generic Functions," "Float Arithmetic Generic Functions," and "String Generic Functions," in Appendix 1 may participate in array expressions with array results. An array may be substituted for any of the arguments of these functions except those arguments which are required to be integer constants, or those which must be converted to integers.

Prefix Operators and Arrays

The result of the operation of a prefix operator or a built-in function upon an array is an array of identical bounds, each element of which is the result of the operation having been performed upon each of the corresponding elements of the original array.

Example:

```
If A is the array          5   3  -9
                           1  -2   7
                           6   3  -4

then -A is the array      -5  -3   9
                          -1   2  -7
                          -6  -3   4
```

Infix Operators and Arrays

Scalar - Array Operations

The result of an operation in which a scalar and an array are connected by an infix operator is an array of bounds identical to the original, each element of which is the result of the operation performed upon the scalar and upon each of the corresponding elements of the original array.

Example:

```
If A is the array          5  10   8
                          12  11   3

then 3*A is the array     15  30  24
                          36  33   9
```

Array - Array Operations

The result of an operation in which two arrays of identical bounds are connected by an infix operator is an array of bounds identical to the original arrays, each element of which is the result of the operation performed upon the corresponding elements of the two original arrays by the infix operator.

Example:

```
If A is the array          2   4
                           3   6
                           1   7
                           4   8

and if B is the array      1   5
                           7   8
                           3   4
                           6   3
```

```
then A + B is the array 3   9
                       10  14
                        4  11
                       10  11

A*B is the array        2  20
                       21  48
                        3  28
                       24  24

and MAX (A+B,A*B) is the array

                        3  20
                       21  48
                        4  28
                       24  24
```

Array Expressions Involving Structures

If an array expression contains structure operands, then all array operands in the expression must be arrays of structures and all involved structures must have the same structuring.

Example:

In the following declaration, A is an array of structures and C is a structure.

```
DECLARE  1A(10),2B,2D,
         1C,2H,2I;
```

Then the expression A+C is a valid expression that will result in the structure C being added to each structure in the array A. The above expression is equivalent to the following:

```
A(1).B + C.H
A(1).D + C.I
A(2).B + C.H
.
.
.
A(10).D + C.I
```

STRUCTURE EXPRESSIONS

The operands of a structure expression are structures, or a combination of structures and scalars. A structure expression returns a structure result. Array operands are not allowed in structure expressions.

All operations performed on structures are performed on an element-by-element basis. Thus, all structures appearing in a structure expression must have identical structuring. This means that the structure must have the same number of contained scalars and arrays. The positioning of the scalars and arrays within the structure must be the same, and arrays similarly positioned must have identical dimensions and bounds. The data types need not be the same.

When an operation has one structure and one scalar operand, it is interpreted as many operations, one for each scalar element in the structure. Each sub-operation involves a structure element and the scalar operand.

A structure expression is a shorthand method of applying an expression to each item of a structure.

Note: A scalar expression is a valid form of a structure expression.

Example:

If there are two structures:

```
1 A                  1 B
2 PART1              2 PART1
  3 SUBPART1           3 SUBPART1
  3 SUBPART2           3 ALPHA
  3 SUBPART3           3 SUBPART2

2 PART2              2 PART2
  3 SUBPART4           3 ALPHA
  3 BETA               3 SUBPART4
  3 SUBPART5 (3)       3 SUBPART5 (3)
```

Then the expression A-2*B is shorthand for the following expressions:

```
A . SUBPART1 - 2*B . SUBPART1
A . SUBPART2 - 2*B . PART1 . ALPHA
A . SUBPART3 - 2*B . SUBPART2
A . SUBPART4 - 2*B . PART2 . ALPHA
A . BETA - 2*B . SUBPART4
A . SUBPART5 - 2*B . SUBPART5
```

Note that the last expression is an array expression.

EVALUATION OF EXPRESSIONS

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

```
¬, **, prefix +, prefix -   highest
*, /                          |
infix +, infix -              |
>=, >, ¬>, ¬=, <, ¬<, <=, =   |
&                             |
|                             |
||                            v
                            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.

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 abnormal functions and abnormal data should be noted (see "Abnormality and Irreducibility," in Chapter 10).

ORDER OF THE EVALUATION OF EXPRESSIONS

The operands of an expression are not accessed in a specific order. A program must not depend on a specific order of access for its successful operation.

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.

CHAPTER 4: DATA DESCRIPTION

ATTRIBUTES

An identifier appearing in a PL/I program may refer to one of many classes of objects. It may, for example, represent a variable referring to a complex number expressed in fixed-point form 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.

Those properties that characterize the object represented by the identifier, and other properties of the identifier itself (such as scope, storage class, etc.), together make up the set of attributes which can be associated with an identifier.

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 identifier by means of a declaration.

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

In a given program, an identifier may represent more than one name. In this case, each different name represented by the identifier is said to be a different use of the identifier. 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 formal parameter list; by this means, an identifier can be established as a name and can be given a certain set (possibly empty) of attributes.

Only one DECLARE statement can be used to establish a given use of a given identifier. However, complementary sets of explicit declarations are permitted:

a. One explicit declaration of an entry name as a statement prefix may be combined with an explicit declaration in a DECLARE statement.

b. 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] name [attribute] ...
[, [level] name [attribute] ...] ...;
```

Syntax rules:

1. Any number of identifiers may be declared as names in one DECLARE statement and must be separated by commas.

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. A blank space is not required to separate a level number from the name following it.

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. 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, and JACK a scalar variable of ten bits.

Factoring of Attributes

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. In the case of a factored level number, the level number precedes the parenthesized list of name declarations.

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

2. DECLARE 1A, 2B, 2B, 2C, 3D, 2D;
   B has been multiply declared.
   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.

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

4. DECLARE 1A, 2A, 3A;
   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 1Y, 2X, 3Z, 3A, 2Y, 3Z, 3A;
   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 ENTRY. If the PROCEDURE or ENTRY statement applies to the outermost procedure of a compilation, the attribute EXTERNAL is given. If all other cases, the attribute INTERNAL is given and the declaration is said to be 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 said to be 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 identifiers appearing in certain contexts to be recognized without an explicit declaration. The various cases are described below.

1. An identifier may occur in a context where only a file name may appear. In some of these cases, the identifier is said to be declared as a file name (see "File Opening and File Attributes" in Chapter 7).

   Example:

   ```
   GET FILE (INFILE) DATA;
   ```

   Here, INFILE is declared contextually with the attribute FILE.

2. An identifier may occur in a context where only a task (or event) name (see "The CALL Statement" in Chapter 8 and "Asynchronous Operations and Tasks" in Chapter 6) may appear. In some of these cases, the identifier is said to be declared as a task (or event) name (see "Application of Default Attributes").

   Example:

   ```
   WAIT (EVENT2);
   ```

   Here, EVENT2 is declared contextually as an event identifier.

3. An identifier may occur in a context where only a programmer-specified condition name (see Appendix 3) may appear. In this case, the identifier is said to be declared as a condition name, with the attribute EXTERNAL.

   Example:

   ```
   ON CONDITION (TEST1) GO TO CHECK;
   ```

   Here, TEST1 is declared contextually as a condition name.

4. An identifier may appear within a statement in a context where only an entry name may appear. That is, an identifier is contextually declared as an entry name if it appears as a label to a PROCEDURE or ENTRY statement or if it appears following the keyword CALL or as the function name in a function reference whose argument list is non-empty. If the occurrence of the identifier does not lie within the scope of the same identifier used to label a PROCEDURE or ENTRY statement, the identifier is given a default attribute of EXTERNAL.

Example:

```
CALL  EXPRI;
```

5. An identifier may appear in a context in which only a pointer name may be used. In this case, the identifier is contextually declared to be a pointer.

   Example:

```
DECLARE A(10,10) CONTROLLED (P);
   ALLOCATE A SET (P);
   P -> A(1,1) = P -> A(5,5);
```

   The variable P is declared contextually as a pointer in each of the above statements.

6. An identifier may appear in a context where only an area name may be used. In this case, the identifier is contextually declared to be an area.

   Example:

```
   ALLOCATE A IN (TREE) SET(P);
```

   In this example TREE is contextually declared to be an area.

Note: Arithmetic or string attributes of constants are determined contextually.

IMPLICIT DECLARATIONS

An identifier may be used in a block without being explicitly declared or contextually declared. In this case the identifier is said to be implicitly declared in the containing external procedure. As will be seen in the discussion of scope, this implicit declaration will then apply to the entire external procedure block except for any contained blocks where the identifier might be explicitly re-declared.

Example:

```
B1:   PROCEDURE (Z1,Z2);
      TEMP1=ABS (Z1**2+Z2**2);
      B2:  BEGIN;
           TEMP2= 1/(TEMP1+Z2)**2;
      IF TEMP2>TEMP1 THEN RETURN
           (TEMP2);
      END B2;
      RETURN (TEMP1);
      END B1;
```

In this example, TEMP1 and TEMP2 are both implicitly declared in block B1.

SCOPE OF DECLARATIONS

When a declaration of an identifier is made in a program, 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.

This definition of scope can be applied to all identifier declarations except the declaration of entry names of external procedures (see "Declarations," in this chapter). The appearance of an identifier as the entry name of an external procedure is regarded as an explicit declaration of the identifier as an entry name with the EXTERNAL attribute. The scope of such a declaration is defined to be the entire external procedure, excluding all contained blocks to which another declaration of the same identifier is internal.

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:

> An explicit or contextual 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 external name is a name that has the scope attribute EXTERNAL. If a name is not external, it is said to be an internal name 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.

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

Reference Line	Name	Use	Scope (by block names)
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

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 E;
       END D;
```

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

Attributes are used to give characteristics to their associated identifiers. The attributes of the language are divided into the following classes:

- Data attributes
- Dimension attribute
- SECONDARY attribute
- REDUCIBLE and IRREDUCIBLE attributes
- ABNORMAL and NORMAL attributes
- USES and SETS attributes
- Entry name attributes
- Scope attributes
- Storage Class attributes
- ALIGNED and PACKED attributes
- DEFINED attribute
- CELL attribute
- INITIAL attribute
- Structure attributes
- LIKE attribute
- File description attributes
- List processing attributes

DATA ATTRIBUTES

Arithmetic Data

Variables are declared to be of arithmetic type if they are given any of the attributes base, scale, mode, or numeric picture.

Base Attributes

Function:
The base attribute specifies that the data is in binary or decimal form.

General format:

```
BINARY|DECIMAL
```

General rules:

These attributes may not be specified in combination with the PICTURE attribute.

Default:

See "Default Conditions for Arithmetic Data" in this chapter.

Examples:

```
DECLARE A DECIMAL, B BINARY;
```

Scale Attributes

Function:

The scale attribute specifies that the data is in fixed-point or floating-point form.

General format:

```
FIXED|FLOAT
```

General rules:

These attributes may not be given in combination with the PICTURE attribute.

Default:

See "Default Conditions for Arithmetic Data."

Examples:

```
DECLARE A FIXED, B FLOAT;
```

Mode Attributes

Function:

The mode attribute specifies that the mode of the data is real or complex.

General format:

```
REAL|COMPLEX
```

General rules:

The COMPLEX attribute may be given in combination with the PICTURE attribute, to specify a complex numeric field.

Default:

See "Default Conditions for Arithmetic Data."

Example:

```
DECLARE A COMPLEX, B REAL;
```

Precision Attribute

Function:

The precision attribute specifies the number of significant binary or decimal digits to be maintained for both fixed-point and floating-point data, as well as the scale of the data.

General format:

```
(number-of-digits[,scale-factor])
```

General rules:

1. The precision attribute must immediately follow a scale, base, or mode attribute at the same factoring level.

2. "Number-of-digits" is a positive decimal integer constant specifying the number of binary or decimal digits to be maintained and is used with both fixed-point and floating-point data.

3. The "scale-factor" is an optionally signed decimal integer constant that defines the position of the point with respect to an integer data item of the specified number of digits. The scale factor is used only with fixed-point data.

4. When the scale is fixed and no scale factor is given, it is assumed to be zero.

5. The scale factor may be negative, and it may be larger than the number of digits.

6. The scale factor effectively multiplies the integer data by the base raised to the power of the scale factor with the sign reversed. For example, decimal data of precision (5,2) represents numbers from .01 to 999.99 or zero in magnitude: decimal data of precision (5,-2) represents numbers from 100 to 9999900 or zero in magnitude.

7. This attribute may not be given in combination with the PICTURE attribute.

Examples:

```
DECLARE A FLOAT (3), B REAL (10)
   FLOAT, X FIXED (5,2);
```

The following table shows the meaning of the scaling for fixed-point variables:

Integer	Scale	Precision	Value
00123	FIXED	(5,2)	1.23
00123	FIXED	(5,-2)	12300
123	FIXED	(3,4)	.0123
123	FIXED	(3,-4)	1230000

Default Conditions for Arithmetic Data

If the base, scale, and mode are not specified, the arithmetic default attributes are dependent upon the first letter of the name. If the first letter of the name is I through N, FIXED REAL BINARY is assumed; otherwise, FLOAT REAL DECIMAL is assumed.

If arithmetic data attributes are partly specified, the remaining attributes are assumed as follows:

```
Base: DECIMAL
Scale: FLOAT
Mode: REAL
```

If precision is not specified, the assumed precision is that which is defined for the particular implementation of the language that is being used, where the definition depends on the scale and base.

The PICTURE Attribute

Function:

The PICTURE attribute is used to define the internal and external formats of numeric and character-string data fields and to specify the editing of data. This discussion is limited to the use of the PICTURE attribute with numeric data. The use of the PICTURE attribute with character-string data is described in "String Attributes." The picture characters are described in Appendix 2.

General format:

```
PICTURE 'numeric-picture-specifica-
   tions'
```

General rules:

1. PICTURE may not be specified in combination with the base, scale, or precision attributes.

   Numeric fields have mode, base, scale, and precision; these are specified by the picture characters used in describing the field, and by the use of the mode attribute if COMPLEX. Note the exception that sterling pictures are treated as a separate category, although they are real fixed-point decimal fields.

2. A "picture specification" is composed of a string of picture characters. It must be enclosed in quotation marks. Individual picture characters may be preceded by an iteration 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 omitted. This iteration factor specification may not follow the picture character F.

3. Numeric picture specifications must include at least one digit position.

4. The following paragraphs indicate the combination of picture characters that show mode, scale, base, and precision. In this discussion, a fixed-point field has one field, and a floating-point field has two subfields.

   a. Real binary fixed-point fields take the following general forms:

   ```
   PICTURE '[S][1] ...  [V]
      [1] ...  [F([+|-] integer) ]'
   PICTURE '[2]...[V][2]...[F([+|-]
      integer)]'
   PICTURE '[3]...[V] [3]...[F([+|-]
      integer)]'
   ```

   Only one V, representing a point, may be present in a picture specification, but it may be in any position. When a sign character (S) is specified, the field will contain a binary 1, if the value is negative, or a zero, if the value is positive.

   b. Real binary floating-point fields take 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]...'
   ```

   The mantissa and exponent must each contain at least one digit position. The sign character allowed to the right of the K in the first form represents the sign of the exponent.

   c. Real decimal fixed-point fields take the following general form:

   ```
   PICTURE '[9]... [V] [9]...
      [F([+|-] integer)]'
   ```

Sign, editing, and zero-suppression picture characters, as explained in Appendix 2, may be included (only one sign character per subfield is allowed). The V may not appear more that once in a picture specification. If no V is given, the decimal point will be assumed to appear to the right of the last digit. No attempt has been made to show the use of all valid picture characters in the general format above. These are explained in Appendix 2.

d. Real decimal floating-point fields take the following general form:

```
PICTURE '[9]...  [V][9]...{E|K}
   9...'
```

The mantissa and exponent must each contain at least one digit position. Sign, editing, and zero-suppression picture characters may be included. Sign characters refer to the subfield in which they appear, except a CR or a DB, which refers to the first subfield. Only one sign character per subfield is allowed.

e. Complex fields may contain those picture characters that are valid for real fields as described above. They take the general form:

```
real-picture
```

The "real-picture" represents both portions of the complex number. The attribute COMPLEX must also be specified. The real-picture may not specify a sterling field.

f. Sterling fields are considered to be real fixed-point decimal fields. When involved in arithmetic operations, they will be 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" may be one or more of the following static picture characters:

```
$ + - S
```

The "pounds field" may contain the following picture characters:

```
Z Y * 9 T I R , $ + - S
```

The last four characters (i.e., $ + - S) must be drifting characters. The comma may be used as a break character.

"Separator 1" may be one or more of the following picture characters:

```
/ . B
```

The "shillings field" may be:

```
{99|YY|ZZ|Y9|Z9|ZY|8}
```

The nines may be replaced by T, I, or R.

"Separator 2" may be one or more of the picture characters:

```
/ . B H
```

The "pence field" takes the form:

```
{99|YY|ZZ|Y9|7|Z9|ZY|6} [V|V.|.V]
[9|Z|Y}...
```

Any of the nines may be replaced by one of the following:

```
T I R
```

"Editing character 2" may be one or more of the static picture characters $ + - S and one or more of B P CR DB.

The pounds, shillings, and pence subfields must each contain at least one digit position.

Zero suppression in sterling pictures is performed on the total field, not separately on each of the pounds, shillings, and pence subfields. In sterling pictures, the subfield separator characters / . B and H are never suppressed.

5. The precision of picture specifications is described below. In this discussion, the following picture characters, actual and conditional, are defined as digit positions:

```
1 2 3 9 Z * Y T I R
  and the drifting
      $ S + -
```

The precision of a fixed-point numeric field is (m,n), where m is the total number of digit positions in the

field and n is the number of digit positions following the V. If a drifting string contains n drifting characters, this specifies n-1 digit positions. For sterling pictures, m is 3 + the number of digits in the pounds field + the number of fractional digits in the pence field.

The precision of a floating-point field is (p), where p is the total number of digit positions before the E or K.

Decimal or binary fixed-point pictures may have a scaling factor. This may be achieved by placing the following at the extreme right of the picture subfield:

```
F ([+|-] integer)
```

with the "integer" value represented by g, this specifies that the decimal or binary point should be assumed to be g places to the right (or left, if negative) of the position assumed in the absence of the scaling factor. The precision of the numeric field is then (m,n-g).

These precisions may not exceed the limits for decimal or binary fixed-point values, as defined for the particular implementation of PL/I.

6. Only one sign position is permitted in a PICTURE subfield. This may be specified by a static sign picture character or by a drifting string for a sign character.

String Attributes

Function:

The string attributes specify string data to be either in bit-string form or in character-string form with a specified length. The form of character-string data may also be specified.

General format:

```
{ { BIT       }(length)  [VARYING] }
{ { CHARACTER }                    }
{                                  }
{ PICTURE 'character-picture-      }
{          specifications'         }
```

General rules:

1. BIT specifies bit-string data, CHARACTER specifies character-string data, and PICTURE specifies character-string data in picture form.

2. The "length" attribute specifies the actual length of fixed-length strings and the maximum length of varying-length strings, in which case the attribute VARYING is given. If VARYING is specified, then either BIT or CHARACTER must also be specified. The attribute VARYING may appear prior to the BIT or CHARACTER attribute in a string attribute specification; that is, it may appear anywhere in the declaration of a string. VARYING may be factored.

3. The length specification may be an expression or an asterisk. It must immediately follow a CHARACTER or BIT attribute at the same factoring level.

4. If the length specification is an expression, it will be converted to an integer at the point of allocation or upon entry to the declaring block for parameters.

5. An asterisk may be used when the length is to be taken from a previous allocation for parameters or nonbased CONTROLLED variables or if it is to be specified in a subsequent ALLOCATE statement for nonbased CONTROLLED variables.

6. The length of strings declared STATIC must be a decimal integer constant.

7. Since PICTURE is an attribute that also may apply to arithmetic data, a separate explanation is in the section entitled "The PICTURE Attribute." Additional picture characters are provided when the PICTURE attribute is used to declare character-string data. These may be found in Appendix 2.

8. BIT, CHARACTER, or VARYING may not be specified if PICTURE is specified.

Example:

```
DECLARE A BIT (10), B CHARACTER (5), C
   PICTURE 'XAA9AA', D BIT(*)VARYING;
```

A is a field of ten bits; B is a field of five characters; C is a field of characters, letters, and a decimal digit; and D is a field of bits with a maximum length to be taken from a previous allocation or to be specified in a subsequent ALLOCATE statement.

The LABEL Attribute

Function:

The LABEL attribute specifies that the associated variable will have statement labels as values. To aid optimization of the object program, it may also specify the values a label variable may have during execution of the program.

General format:

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

General rules:

1. If the variable is a parameter, the value can also be any statement label that could be passed as an argument, or any value permitted for any label variable that may be specified as an argument.

2. If a list of statement-label constants is specified, the variable may have as values only members of the list. The label constants in the list must be known in the block containing the declaration.

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

4. A subscripted label that is an element of a label array may appear as a statement prefix but may not appear in an END statement after the keyword END.

Example:

```
DECLARE START LABEL (LABEL1, LABEL2,
   LABEL3);
```

The TASK Attribute

Function:

The TASK attribute specifies that the associated identifier is used as a task name (see "Asynchronous Operations and Tasks," in Chapter 6, the general rules under "The CALL Statement," in Chapter 8, and "Task Data" in Chapter 2).

General format:

```
TASK
```

General rules:

1. An identifier may be explicitly declared with the TASK attribute in a DECLARE statement. It may be contextually declared by its appearance in a TASK option appended to a CALL statement (see Chapter 8).

2. Task names may also have the following attributes:

   Dimension attribute
   Scope attribute (the default is INTERNAL)
   Storage class attribute (the default is AUTOMATIC)
   DEFINED attribute (task names may only be defined on other task names)
   ABNORMAL attribute (all task names are ABNORMAL)
   SECONDARY attribute

3. A task name can appear in a TASK option (see "The CALL Statement," in Chapter 8), as the argument in the PRIORITY built-in function, or in the PRIORITY pseudo-variable. Task names also may be passed as procedure parameters.

The EVENT Attribute

Function:

The EVENT attribute specifies that the associated identifier is used as an event name (see "Asynchronous Operations and Tasks," in Chapter 6, the general rules under "The CALL Statement," in Chapter 8, and "Event Data" in Chapter 2).

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 appended to a CALL statement, in a WAIT statement, in a DISPLAY statement, or in various input/output statements (see Chapter 8).

2. Event names may also have the following attributes:

   Dimension attribute
   Scope attribute (the default is INTERNAL)
   Storage class attribute (the default is AUTOMATIC)
   DEFINED attribute (event names may

only be defined on other event names)
ABNORMAL attribute (all event names are ABNORMAL)
SECONDARY attribute

3. An event name can appear in an EVENT option, a WAIT statement (see Chapter 8), or as the argument in the EVENT built-in function or in the EVENT pseudo-variable. Event names also may be passed as procedure parameters.

THE DIMENSION ATTRIBUTE

Function:

The dimension attribute defines the bounds of an array.

General format:

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

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

Syntax rule:

Lower bound and upper bound are scalar expressions.

General rules:

1. The number of "bounds" specifies the number of dimensions in an array.

2. Bounds that are expressions are evaluated and converted to integer data when storage is allocated for the array or when linkage is established for parameters.

3. The bounds are indicated as follows:

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

   b. When the actual bounds for each dimension are to be taken from a previous allocation for that identifier or are to be specified in a subsequent ALLOCATE statement for nonbased variables, an asterisk must be used to represent each of the dimension bounds. Thus, asterisks may be used only for parameters and CONTROLLED variables.

   c. The lower bound must be less than or equal to the upper bound.

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

5. If an attribute list contains a dimension attribute, that attribute must come first in the list.

6. If any bound of a dimension attribute in a structure declaration is an asterisk, then all dimension bounds for the major structure and for all other structure elements must also be asterisks.

7. The asterisk notation may not be used for based variables.

Examples:

1. DECLARE TABLEA(5,8), TABLEB(-5:5,10);

   TABLEA is a two-dimensional array with 5 rows and 8 columns (subscripts 1 to 5 and 1 to 8). TABLEB is a two-dimensional array with 11 rows and 10 columns (subscripts -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 for the rows and 1 through 10 for the columns).

2. DECLARE MATRIX (*,*);

   MATRIX is a two-dimensional array. The bounds are to be taken from a previous allocation for MATRIX or are to be subsequently specified in an ALLOCATE statement.

THE 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.

THE ABNORMAL AND NORMAL ATTRIBUTES

Function:

The ABNORMAL and NORMAL attributes are used to specify data as being either normal or abnormal.

General format:

ABNORMAL|NORMAL

General rules:

1. The ABNORMAL attribute may be declared for any variable.

2. The ABNORMAL attribute specifies that a variable may be altered or otherwise accessed at an unpredictable time during the execution of a program. This situation might occur, for example, during the execution of an ON-unit as described in "The ON Statement," in Chapter 8.

3. Every time ABNORMAL data is referred to, its associated storage contains its current value.

Default for Abnormality of Data

Variables are assumed to be NORMAL, except structures containing ABNORMAL elements; such structures may not be declared NORMAL.

THE REDUCIBLE AND IRREDUCIBLE ATTRIBUTES

Function:

The REDUCIBLE and IRREDUCIBLE attributes are used to specify entry names as being either reducible or irreducible. The IRREDUCIBLE attribute specifies that invocations of the specified entry may not not be reduced to a smaller number of invocations.

General format:

REDUCIBLE|IRREDUCIBLE

General rules:

1. Reducibility is a property of both external and internal procedures. Blocks invoking procedures that are irreducible must be within the scope of an IRREDUCIBLE, USES, or SETS declaration for the invoked entry name. However, the invocation of an irreducible procedure does not make the invoking procedure itself irreducible. These attributes enable program optimization to be performed.

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

   a. Access, modify, allocate 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 above for external procedures.

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

4. Irreducible external procedures invoked as functions must be declared with at least one of the attributes, IRREDUCIBLE, USES, or SETS. The scope of this declaration must include the invoking block.

5. IRREDUCIBLE used alone specifies that all possible types of irreducibility should be assumed. It is unnecessary to specify IRREDUCIBLE for the built-in functions, TIME and DATE.

6. The REDUCIBLE attribute specifies that the entry name is for a procedure that is not irreducible.

Default for Irreducibility of Procedures

If an external entry name appears only as a function reference, the entry name is assumed to have the REDUCIBLE attribute. Entry names of all internal procedures and entry names of external procedures invoked in CALL statements are assumed to have the IRREDUCIBLE attribute.

THE USES AND SETS ATTRIBUTES

Function:

The USES and SETS attributes are used to specify, for an entry name, the nature of its irreducibility due to data manipulation.

General format:

```
USES (item[,item]...)
SETS (item[,item]...)
```

General rules:

1. The items of the list following a USES or SETS attribute may be as follows:

   a. A decimal integer n, specifying the nth argument of any invocation of the procedure at the declared entry name.

   b. An unsubscripted data name known to both the block containing the declaration and the invoked procedure.

   c. An asterisk indicating all identifiers described in b.

2. An item in the USES list specifies the following:

   a. That the invoked procedure or procedures invoked by it access that item.

   b. That neither the invoked procedure nor procedures invoked by it reassign that item unless it is also specified in a SETS attribute.

   c. That neither the invoked procedure nor procedures invoked by it access any other data known to the block, except data designated by explicit arguments in either a CALL statement, a statement with a CALL option, or a function reference.

3. An item in the SETS list specifies the following:

   a. That the invoked procedure or procedures invoked by it reassign, allocate, or free that item.

   b. That neither the invoked procedure nor procedures invoked by it 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. That neither the invoked procedure nor procedures invoked by it reassign, allocate, or free any other data known in the block.

4. 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.

5. If an item in a USES or SETS list, as described in 1b above is defined on a base (see "The DEFINED Attribute") 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.

6. A structure or array name appearing in a USES or SETS list implies that the names of all items contained in the structure or array also are on the list. It does not imply that items defined on elements of the structure are in the list; these must be declared as in rule 5, above.

7. If the USES or SETS attribute is specified and the invoked procedure is irreducible in any other way, the IRREDUCIBLE attribute must still be specified (unless it is given by default). If the USES or SETS attribute is specified and the invoked procedure is not otherwise irreducible, the IRREDUCIBLE attribute should not be specified.

ENTRY NAME ATTRIBUTES

An identifier may be declared to be an entry name by giving it the ENTRY attribute. It may be declared to have any of the attributes SETS, USES, BUILTIN, and RETURNS. These attributes all imply ENTRY and thus ENTRY need not be specified. The entry name also may have the attributes IRREDUCIBLE or REDUCIBLE.

An explicit declaration of an internal entry name and the procedure block having the entry name must both be internal to the same block.

An identifier may be declared as representing a family of entry names, by using the GENERIC attribute.

The ENTRY Attribute

Function:

The ENTRY attribute is used to declare, within a procedure, entry names that are referred to in that procedure.

General format:

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

General rules:

1. When ENTRY is used, it specifies that the identifier being declared is an entry name. An entry name must be declared with the ENTRY attribute unless the entry label is known in the same block, or unless a reference is made to the entry name in a CALL statement or in a function reference with arguments, or if it is declared to have any of the attributes SETS, USES, GENERIC, BUILTIN, and RETURNS. INTERNAL entries may only be declared in the block to which the procedure is internal. ENTRY without a parameter attribute list specifies nothing about the number or nature of the parameters.

2. When ENTRY is used with parameter attribute lists, each parameter attribute list is a succession of attributes describing a parameter of the entry point. Permitted attributes are those allowed for parameters.

3. The number of parameter attribute lists must be the same as the number of parameters required by the entry point. If a parameter attribute list is null, its place must be kept by a comma.

4. Parameter attribute lists are not necessary if the parameters of the entry name are not to be described.

5. The dimension attribute may be specified for array parameters. It must be the first attribute specified for the parameter.

6. The structuring for a structure parameter is specified by a structure description using level numbers without identifiers, the level number being immediately followed by the list of attributes for that level of the structure. The first item in the description of the structure parameter must be at level one.

7. Expressions occurring in ENTRY attributes for length or dimension bounds are evaluated upon entering the block to which the declaration of the ENTRY attribute is internal. If an argument position specifies an entry with no data attributes, no default data attributes are provided.

Default:

If no attributes or level numbers are given for a parameter, no assumptions are made about it. When any attributes are specified, the remaining required attributes are deduced according to the default rules given in "Assignment of Attributes to Identifiers." Note that if the partially specified attributes imply data elements without specifying the type, arithmetic REAL FLOAT DECIMAL is assumed.

The GENERIC Attribute

Function:

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 may 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. Each entry name declaration must have the ENTRY attribute. It may optionally have IRREDUCIBLE, REDUCIBLE, USES, SETS, and RETURNS attributes. No entry name declaration may have the GENERIC attribute.

4. Each entry name declaration must specify attributes or level numbers for every parameter of the associated entry name. Attributes unspecified but required for full definition will be deduced from default rules.

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 and only one 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.

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 range list)
   Dimensionality (but not bounds)
   CHARACTER (but not length)
   BIT (but not length)
   VARYING
   TASK
   EVENT
   POINTER
   AREA
   ENTRY (but not parameter description or other attributes of entry names)
   FILE (but no other FILE attributes)
   Structuring, including only the attributes listed above for the structure members.

   If precision is specified by FLOAT (*), then the precision is not taken into account in the matching process.

7. Generic entry names (as opposed to references) may be specified as arguments to non-generic procedures if the invoked entry name is declared with the ENTRY attribute (explicit or implicit for internal procedures). This ENTRY attribute must specify that the appropriate parameter is an entry name and 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.

Example:

```
DECLARE
  CALCULATE GENERIC (FIXCALC ENTRY (FIXED),
  FLTCALC ENTRY (FLOAT)), Y FLOAT
  INITIAL (50);
  X=Y + CALCULATE (Y);
```

The assignment statement results in the invocation of the procedure FLTCALC, since the argument Y matches the entry attribute of the FLTCALC member of the family.

The BUILTIN Attribute

Function:

The BUILTIN attribute specifies that the reference to the associated identifier within the scope of the declaration is 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 this name has been declared to have another use.

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

3. The BUILTIN attribute may not be declared for formal parameters.

For a list of built-in functions see Appendix 1.

The RETURNS Attribute

Function:

The RETURNS attribute is specified with an explicitly declared entry name in order to define the data attributes of the value to be returned by that entry.

General Format:

```
RETURNS [(attribute ...)]
```

General Rule:

The attributes specify the data characteristics of the value returned by the entry when it is invoked as a function. If data attributes are not specified, defaults will be applied (see "Assignment of Attributes to Identifiers" in this chapter). Only string, arithmetic, and pointer attributes may be specified. Note that the attributes of the value returned by the function should agree with the attributes specified with RETURNS; if they do not

agree, it is an error since no conversion will be performed.

SCOPE ATTRIBUTES

Function:

The scope attributes are used to specify the scopes in which declared identifiers are known.

General format:

```
{ INTERNAL }
{ EXTERNAL }
```

For a full discussion of the INTERNAL and EXTERNAL attributes, see "Scope of Declarations".

Default:

If the scope is unspecified for variable names, INTERNAL is assumed.

STORAGE CLASS ATTRIBUTES

Function:

Storage class attributes are used to allocate and/or describe a particular class of storage to variables.

General format:

```
STATIC|AUTOMATIC|CONTROLLED|CONTROLLED
   (pointer-variable)
```

General rules:

1. STATIC specifies that storage is allocated at the start of execution of the program and is not released until program execution has been completed.

2. AUTOMATIC specifies that storage is allocated on each entry to the block to which the storage declaration is internal. The storage is released on leaving the block. If the block is a procedure that is invoked recursively, the previously allocated storage is "pushed down" on entry, and the latest allocation of storage is "popped up" on termination. (For a discussion of "pushed down" and "popped up" storage, see "Allocation of Data and Storage Classes" in Chapter 6.)

3. CONTROLLED specifies that full control will be maintained over the allocation and freeing of storage by means of the statements ALLOCATE and FREE.

4. AUTOMATIC variables may have INTERNAL scope only. STATIC and CONTROLLED variables may have INTERNAL or EXTERNAL scope.

5. Storage class attributes may not be specified for entry names, file names, members of structures, or DEFINED data.

6. STATIC and AUTOMATIC attributes may not be specified for parameters.

7. Variables declared with adjustable lengths and dimensions may not have the STATIC attribute.

8. If a procedure involving static storage is invoked from within or as a separate task, the static storage is common to all invocations.

9. If, during execution of a statement, controlled data is allocated or freed (by an irreducible function, for example), any reference in the statement to that data produces an undefined result.

10. Storage class attributes may only be given for variables at level 1. The storage class applies to all elements of a structure or array of structures. If a structure is controlled, only the major structure, and not the elements, may be allocated and freed.

11. The CONTROLLED (pointer-variable) attribute is used in connection with list processing and RECORD transmission. The variable declared with this form of the attribute is called a based variable. The following rules govern the use of pointer and based variables with the CONTROLLED (pointer-variable) attribute.

    a. The pointer variable may be given additional attributes, but such attributes must be declared separately. If additional attributes are not declared, the default attribute AUTOMATIC applies.

    b. When reference is made to a based variable, the data attributes assumed are those of the based variable, while the associated pointer variable identifies the generation of data. If the reference is to a component of a based structure, a second, temporary pointer variable is created to determine the location of the component in relation to the beginning of the structure (that is, the offset of the component within the structure).

c. Array dimensions 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.

d. A based variable may be used to identify and describe data existing in any storage class, or to obtain storage (via the ALLOCATE statement) which has the characteristics of the based variable.

e. The scope of a based variable is internal to the block in which it is declared; therefore, the attribute EXTERNAL may not appear with a based variable declaration.

f. The attribute VARYING may not be specified for a based variable.

g. The INITIAL attribute may be specified for based variables. The values are assigned only upon explicit allocation of the based variable in an ALLOCATE statement.

h. Based variables may not be specified in the CHECK condition.

i. When a based variable incorporating arrays or character strings is an argument for a procedure invocation, its dimensions and/or lengths are evaluated and then fixed for the duration of the invocation.

Default:

1. If storage class is unspecified and the scope is EXTERNAL, STATIC is assumed.

2. If storage class is unspecified and the scope is INTERNAL, AUTOMATIC is assumed.

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

Examples:

1.
```
EXAMPLE: PROCEDURE;
         DECLARE A STATIC INITIAL
           (0), B CONTROLLED, C(10);
         ALLOCATE B;
         A = A + 1;
             .
             .
             .
         FREE B;
         PUT LIST(A);
         END EXAMPLE;
```

The variable A is of the static storage class and is used to count the number of times the procedure is invoked. The variable B is of the controlled storage class, and storage is allocated and freed by use of the ALLOCATE and FREE statements. The variable C is of the automatic storage class by default.

2.
```
DECLARE VALUE CONTROLLED (P);
```

The variable VALUE is a based variable in which the pointer P is used to locate the generation of VALUE when reference is made to it. The scope of VALUE is internal, and the pointer variable P is of the automatic storage class by default.

3.
```
DECLARE STRINGS (I,J) CHARACTER (K)

        CONTROLLED (Q),

        Q STATIC EXTERNAL;
```

The variable STRINGS is an array of character strings based upon the pointer Q. The values of I and J will be evaluated dynamically at each reference to STRINGS to determine the dimensions of STRINGS, and the value K will be dynamically evaluated to determine the length of each element. The pointer variable Q will appear in static external storage.

THE ALIGNED AND PACKED ATTRIBUTES

Function:

The ALIGNED and PACKED attributes are used to specify in storage the arrangement of string or numeric field data elements within data aggregates.

General format:

```
ALIGNED|PACKED
```

General rules:

1. These attributes may be specified for the following:

   a. Names of major structures.

   b. Names of arrays that are not themselves part of a structure.

2. PACKED specifies that each string or numeric field element is packed in storage contiguous with the string or numeric field elements that surround it. There should be no unused storage between two adjacent elements, provided all data elements of the aggregates are string or numeric field variables of the same type. In other cases, some unused space may appear but storage is to be conserved when possible.

3. ALIGNED specifies that each string data element within the aggregate may start at a storage boundary to be defined individually for each implementation of PL/I. This implies that two adjacent string or numerical field elements of a homogeneous aggregate may not necessarily occupy contiguous storage, if a more efficient program is possible.

4. Arguments to the STRING generic function must be PACKED structures.

Default:

1. The default for major structures is PACKED.

2. The default for arrays that are not part of structures is ALIGNED.

Examples:

```
DECLARE
    1 A (10)  PACKED, 2 B BIT
    (200), 2 C BIT (500), 2 D BIT
    (300), E (10,15) ALIGNED BIT (15);
```

All elements of A, an array of structures, will occupy a continuous area of storage. Each element of the array E will start at a storage boundary defined for that implementation of PL/I. There may be unused storage between the elements of the latter array.

THE DEFINED ATTRIBUTE

Function:

The DEFINED attribute specifies that scalar, array, or structure data is to occupy the same storage area as that assigned to other data.

General format:

DEFINED base-identifier [subscript list]

Rules for defining:

1. The INITIAL, the storage class, and the scope attributes must not be specified for the defined item. The VARYING attribute must not be specified for either the defined item or the base identifier. It should be noted that although the base may have the EXTERNAL attribute, the defined item always has the INTERNAL attribute. If the base is declared external, its name will be known in all blocks in which it is declared external, but the name of the defined item will not. However, the value of the defined item will be changed if the value of the base item is changed in an external block.

2. The base identifier must always be known within the block where the defined item has been declared; the base identifier must not have the DEFINED attribute, nor may it be a based variable.

There are two types of defining, correspondence defining and overlay defining.

If iSUB variables are involved, or if both the defined item and base identifier are arrays with the same number of dimensions and the POSITION attribute is not specified, correspondence defining is in effect. In all other cases, overlay defining is in effect.

In correspondence defining, the elements of the base identifier and the elements of the defined item must have the same description.

Correspondence Defining

When correspondence defining has been specified, a reference to an element of the defined item is interpreted as a reference to the corresponding element of the base identifier. A reference to the defined array is interpreted as a reference to the aggregate of all of the base elements that correspond to some element of the defined array. Note that the base array must not be a cross section of a larger aggregate.

If there is no subscript list following the base identifier, then the correspondence is direct. In such a case, the arrays must have the same number of dimensions, and a reference to an element of the defined item would be interpreted as a reference to an element of the base with the same subscripts.

If a subscript list follows the base identifier, each subscript may be an expression and each expression may contain references to the dummy variables indicated by iSUB.

In the dummy variable iSUB, i is a decimal integer constant in the range 1 to n, where n is the dimension of the defined item.

At least one reference to iSUB must appear in the subscript list. An array defined by using iSUB variables in the subscript list cannot be passed as an argument.

The base element corresponding to a defined element is obtained by replacing each iSUB in the subscript list by the integer value of the ith subscript of the defined element.

Reference may not be made to any element of the defined item that does not have a corresponding element in the base identifier.

Overlay Defining

Overlay defining specifies that the defined item is to occupy part or all of the storage allocated to the base. In this way, changes to the value of either variable may be reflected in the value of the other. Overlay defining is permitted between the following:

	Defined Item	Base Identifier
1.	A scalar coded arithmetic variable	A subscripted or unsubscripted coded arithmetic scalar of the same base, scale, mode, and precision
2.	A scalar label variable	A subscripted or unsubscripted scalar label variable
3.	A scalar pointer variable	A subscripted or unsubscripted scalar pointer variable
4.	An area variable	A scalar area variable
5.	A scalar event variable	A subscripted or unsubscripted scalar event variable
6.	A scalar task variable	A subscripted or unsubscripted scalar task variable
7.	A bit class variable	Bit class data that is not a cross section either of an array or of an array within an array of structures

Note: The bit class consists of:

a. Numeric binary fields

b. Fixed-length bit strings

c. Packed structures consisting of items a, b, and d

d. Packed arrays consisting of items a, b, and c

8.	A character class variable	Character class data that is not a cross section either of an array or of an array within an array of structures

Note: The character class consists of:

a. Numeric picture fields

b. Fixed-length character strings

c. Packed structures consisting of items a, b, and d

d. Packed arrays consisting of items a, b, and c

9.	A structure	An identical structure whose makeup is such that matching pairs of items from the structures are valid examples for overlay defining of the types described in items 1 through 6 above

Rules for overlay defining:

1. In items 7 and 8 above, the POSITION attribute may be specified for the defined item. If POSITION is specified, the DEFINED attribute must also be specified. POSITION need not necessarily follow the appearance of DEFINED; it may precede it in the same declaration, if so desired. The general format of the POSITION attribute is as follows:

POSITION (decimal-integer-constant)

This specifies the position, in relation to the start of the base, at which the defined item is to begin. If this attribute is omitted, POSITION (1) is assumed; i.e., the defined item is to begin at the first position of the base.

2. In items 7 and 8 above, the extent of the defined item must not be larger than the extent of the base. Extent is calculated by summing the lengths of the parts of the data, including all individual elements of arrays, and, in the case of the defined item, adding n-1 (where n is the position in relation to the start of the base).

Order of Evaluation

Evaluation proceeds as follows:

1. Expressions specified in all attributes of the defined item (other than the DEFINED attribute) are evaluated on entry to the declaring block.

2. Subscripts of the base identifier are evaluated when a reference to the defined item is made.

3. Data defined on a CONTROLLED base normally refers to the most recent generation of base data. However, if a defined item appears as an argument to an invoked procedure, and the base is reallocated, the value of the argument will be based on the generation current at the time of invocation.

Examples of Defining

1. DECLARE A(20,20), B(10) DEFINED A(2*1SUB, 2*1SUB);

In the first example, B is a vector consisting of every even element in the diagonal of matrix A. In other words, B(1) corresponds to A(2,2), B(2) corresponds to A(4,4), etc.

2. DECLARE 1 P, 2 Q CHARACTER (10),
2 R CHARACTER (100),
PSTRING1 CHARACTER (110)
DEFINED P;

3. DECLARE LIST CHARACTER (40),
ALIST CHARACTER (10) DEFINED LIST,
BLIST CHARACTER (20) DEFINED LIST POSITION (21),
CLIST CHARACTER (10) DEFINED LIST POSITION (11);

In the third example, ALIST corresponds to the first ten characters of LIST, BLIST corresponds to the twenty-first through fortieth characters of LIST, and CLIST corresponds to the eleventh through twentieth characters of LIST.

THE CELL ATTRIBUTE

Function:

The CELL attribute establishes the associated identifier as a cell and specifies that each alternative 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 at least two data declarations.

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 identifier itself may appear only in DECLARE, ALLOCATE, and FREE statements, as well as in the context of arguments and parameters.

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 A 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 (8,3),
               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.

THE INITIAL ATTRIBUTE

Function:

The INITIAL attribute either specifies constant values to be assigned to data when storage is allocated to it, or it specifies, through the CALL option, a procedure to be invoked to perform initialization at allocation.

General format:

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

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

Rules for form 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 may be specified for a scalar; more may be given for an array.

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 may 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)*

7. The "iteration factor" may be any expression that satisfies the rules stated in the section on "Prologues" in Chapter 10. When storage is allocated, the expression is evaluated to give an integer that specifies the number of repetitions.

8. Only unsigned decimal integer constants are permissible as iteration factors for STATIC data.

9. A negative or zero iteration factor yields no initialization.

10. Iterations may be nested.

11. Label constants given as initial values for label variables must be known within the block in which the label variable declarations occur.

12. An alternate 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. (Such a statement prefix may not be pointer qualified.) The effect of this appearance is the initialization of that array element to a constructed label constant for the statement carrying the subscripted reference. This statement must be internal to the block containing the declaration of the array. Only one form of initialization may be used for a given label array. (See the sixth example at the end of this section for an illustration.)

13. The INITIAL attribute may not be given for the following:

    entry names
    file names
    DEFINED data
    structures
    parameters
    TASK data
    EVENT data
    AREA data

    Notes: The INITIAL attribute may be given for base elements of structures. General rule 13 also applies to form 2.

14. If only one parenthesized scalar expression precedes a string initial value, it is interpreted as a replication factor for the string. If two appear, the first is taken to be an initialization iteration factor, the second, a string replication factor. For example:

    ((2)'A') is equivalent to ('AA')
    ((2)(1)'A') is equivalent to ('A','A')

Rules for form 2:

1. The entry name and arguments passed must satisfy the conditions stated in "Prologues."

2. This form may not be used to initialize STATIC data.

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 PTS(5) POINTER INITIAL
        ((5)NULL);
```

6.
```
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.

THE LIKE ATTRIBUTE

Function:

The LIKE attribute specifies that the name being declared is given the same structuring as the name following the attribute LIKE.

General format:

```
LIKE structure-name
```

General rules:

1. The "structure name" may be unqualified or qualified, but it may not be subscripted.

2. The structure must be known to the block containing the LIKE attribute.

3. Neither the structure name nor any of its substructures can be declared with the LIKE attribute.

4. The LIKE attribute specifies that the name being declared is a structure with a substructure having elements with attributes and names identical to the names and attributes of the elements of the named structure. Contained dimension and length attributes are recomputed. Attributes of the structure name itself do not carry over, only its elements enter into this process.

5. If the structure description of the named structure has been declared, and if a direct application of the description to the structure being declared LIKE would cause an incorrect discontinuity in level numbers, then the level numbers will be modified by a constant before application.

6. The number that immediately follows the member that has the LIKE attribute must be a level-number that is equal to or less than that of the member that has the LIKE attribute.

Examples:

```
1.  DECLARE 1 A(10),
            2 FIELD1,
              3 DTL1 PIC'$ZZ.99',
              3 DTL2 CHAR (10),
            2 FIELD2 BIT (50),
          1 X,
            2 FIELD1,
              3 SUBFLD1 (20) LIKE A.FIELD1,
              3 TABLE (3),
            2 FIELD2 LIKE A . FIELD1;
```

The above is equivalent to:

```
 DECLARE 1 A(10),
            2 FIELD1,
              3 DTL1 PIC '$ZZ.99',
              3 DTL2 CHAR (10),
            2 FIELD2 BIT (50),
          1 X,
            2 FIELD1,
              3 SUBFLD1 (20),
                4 DTL1 PIC '$ZZ.99',
                4 DTL2 CHAR (10),
              3 TABLE (3),
            2 FIELD2,
              3 DTL1 PIC '$ZZ.99',
              3 DTL2 CHAR (10);
```

```
2.  DECLARE 1 A EXTERNAL, 2(B,C,D), 1 E
        . LIKE A;
```

The above is equivalent to :

```
 DECLARE 1 A EXTERNAL, 2(B,C,D), 1 E,
       2(B,C,D);
```

FILE DESCRIPTION ATTRIBUTES

File description attributes are used to describe data files. Declarations of the same file in more than one external procedure must not conflict (for a complete discussion of data files and the default attributes, see Chapter 7).

The FILE Attribute

Function:

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

General format:

```
FILE
```

Note that the FILE attribute is implied by every one of the file description attributes described in this section and thus need not be specified in a context in which at least one of these attributes is given for a filename. However, if such a context contained only an INTERNAL or EXTERNAL attribute, FILE would have to be specified to establish the filename.

The File Usage Attributes

Function:

The file usage attributes specify the method of treatment of data in the file.

General format:

STREAM|RECORD

Rules:

1. A file with the STREAM attribute may be used only in the OPEN, CLOSE, GET, and PUT statements. A file with the RECORD attribute may be used only in the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, DELETE, and UNLOCK statements.

2. A file with the STREAM attribute cannot have any of the following attributes: RECORD, UPDATE, DIRECT, SEQUENTIAL, BACKWARDS, BUFFERED, UNBUFFERED, EXCLUSIVE, KEYED.

The Function Attributes

Function:

The function attributes specify the function of a file.

General format:

INPUT|OUTPUT|UPDATE

Rules:

1. INPUT specifies that the data will be transmitted only from the data set to the program. A file with the INPUT attribute cannot have the attributes EXCLUSIVE or PRINT.

2. OUTPUT specifies that the data will be transmitted only from the program to the data set. A file with the OUTPUT attribute cannot have the attributes EXCLUSIVE or BACKWARDS.

3. UPDATE specifies that the file is to be used for both input and output. A declaration of UPDATE for a file with SEQUENTIAL access denotes the update-in-place mode. Such files must be accessed in the sequence READ, then REWRITE. A file with the UPDATE attribute cannot have the attributes STREAM, BACKWARDS, or PRINT.

The PRINT Attribute

Function:

The PRINT attribute specifies that the ultimate disposition of the data is to be the printed page. Several special options are permitted on PUT statements that refer to files having the PRINT attribute.

General format:

PRINT

Rules:

1. A file with the PRINT attribute implies the OUTPUT and STREAM attributes.

2. This attribute cannot be specified for a RECORD file.

The Access Attributes

Function:

The access attributes specify the manner in which the records within a RECORD file are accessed.

General format:

SEQUENTIAL|DIRECT

Rules:

1. If a file is DIRECT, each record transmission must specify a key. A record written with a particular key can be retrieved by reading with that value of key specified. Files with the DIRECT attribute must also have the KEYED attribute.

2. SEQUENTIAL normally specifies that the next record to be accessed is determined by the physical organization of the data set.

The Buffering Attributes

Function:

The buffering attributes apply to SEQUENTIAL RECORD files only, and specify whether or not the records must pass through intermediate storage during transmission to and from the data set. If there is such buffering, the intermediate storage can be accessed by associating it with a pointer variable, and using the pointer to identify a based variable that describes the record in the buffer (see the discussion of "RECORD Transmission" in Chapter 7).

General format:

```
BUFFERED|UNBUFFERED
```

General rule:

A file with STREAM or DIRECT attributes cannot have a buffering attribute.

The BACKWARDS Attribute

Function:

The BACKWARDS attribute specifies that a SEQUENTIAL INPUT file is to be accessed in reverse order, i.e., from the last member to the first member.

General format:

```
BACKWARDS
```

The EXCLUSIVE Attribute

Function:

The EXCLUSIVE attribute specifies that a DIRECT UPDATE file will be used in such a way as to prevent one task reading, deleting, or rewriting a record while another task is in the process of reading, deleting, or rewriting that record (see "The READ Statement," in Chapter 8).

General format:

```
EXCLUSIVE
```

The ENVIRONMENT Attribute

Function:

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

General format:

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

General rules:

1. The option list will be defined individually for each implementation of PL/I.
2. The options must be separated by one or more blanks.

The KEYED Attribute

Function:

The KEYED attribute specifies that each record in the file has a key associated with it.

General format:

```
KEYED (decimal-integer-constant)
```

General rules:

1. A KEYED file cannot have the attributes STREAM or PRINT.
2. The "decimal integer constant" gives the length of the key in characters.
3. The KEYED attribute must be specified for every file containing keys, even if records are read sequentially.

LIST PROCESSING ATTRIBUTES

The AREA Attribute

Function:

The AREA attribute is used to define an area of storage which may be used for collecting and referring to based data items.

General format:

Option 1:

AREA

Option 2:

AREA (d_1, d_2, ..., d_n)

(where each d represents a data declaration without identifiers)

General rules:

1. An area variable may be explicitly declared with the AREA attribute in a DECLARE statement. It may be declared contextually by its appearance in the IN clause of an ALLOCATE statement. A contextual declaration implies that Option 1 will be used.

2. Option 1 specifies that an implementation-defined amount of storage will be allocated for the area variable.

3. Option 2 provides programmer control of the amount of storage allocated for an area variable. The data declarations in this option are dummy declarations; their sole purpose is to specify the amount of storage to be allocated. It is not required or expected that the data variables actually allocated into the storage area will match the data declarations in number, order, or attributes. However, if the allocations do not conform to the attributes and their order, there may not be sufficient storage to contain all allocations.

4. The individual data declarations in Option 2 are similar to parameter descriptions of an ENTRY attribute. Since they are dummy declarations, they may not specify identifiers. If dimensions are given in the declaration, they must appear first.

5. Area variables are not valid operands for any operators in the language, including assignment. Conversions to and from area variables are not defined.

6. Area variables may not be transmitted in input/output operations.

7. Area variables may not appear in the CHECK condition.

8. Area variables may be elements of arrays and components of structures.

9. Entry points may not return a value of type area.

Example 1:

```
DECLARE TABLE_1 AREA STATIC EXTERNAL;
```

TABLE_1 is a static external area of implementation-defined size.

Example 2:

```
DECLARE TABLE_2 AUTOMATIC
AREA ((100) POINTER, (50) CHARACTER
(30), 1(50), 2 FIXED, 2 POINTER);
```

TABLE_2 is an automatic area that is large enough to contain an array of 100 pointers, an array of 50 character strings of length 30, and an array of 50 structures, each consisting of a fixed-point value followed by a pointer. However, the area need not be used in this way (see Rule 4 above).

The POINTER Attribute

Function:

The POINTER attribute specifies that the associated identifier may be used to identify data values existing in any storage class.

General format:

POINTER

General rules:

1. An identifier may be explicitly declared with the POINTER attribute in a DECLARE statement. It may be contextually declared by (a) its appearance with a CONTROLLED attribute, (b) its appearance in the SET option of an ALLOCATE, READ, or LOCATE statement, or (c) its use as a pointer qualifier.

2. The value of a pointer may be established by (a) assignment, (b) the SET clause in an ALLOCATE, LOCATE, or READ statement, or (c) use of the INITIAL attribute.

3. Pointer data may not be used directly as an operand in an arithmetic expression, nor may conversions be performed between pointer data and other data types.

4. The only operators that may be applied directly to pointer data are the comparison operators = and ¬=.

5. Pointer data may not be read or written in STREAM input/output.

6. Pointers may be initialized only to the NULL value or to the value of another pointer variable.

7. Entry points may return a value the data type of which is pointer.

Examples:

1. DECLARE P POINTER STATIC;
   The pointer P is declared explicitly.

2. DECLARE VALUES CONTROLLED (PT1);
   The pointer PT1 is declared contextually. It will reside in the AUTOMATIC storage class by default.

3. ALLOCATE VALUES SET (PT3);
   The data type of PT3 is pointer by contextual declaration in the SET clause.

ASSIGNMENT OF ATTRIBUTES TO IDENTIFIERS

Identifiers 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.

Within an external procedure, statement label constants, internal entry labels, parameters, and identifiers appearing in DECLARE statements are qualified by the respective blocks in which their declarations (contextual or explicit) occur. Thus they serve as a means of redeclaring identifiers declared explicitly, contextually, or implicitly in containing blocks. For an identifier occurring as a parameter, the characteristic, "parameter," is combined with any explicitly declared attributes for the identifier. Default attributes are added as described below. 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.

The following attributes, assigned through context, are recognized in the indicated ways:

1. ENTRY (subroutine): CALL statement or CALL option

2. ENTRY (function): identifier followed by parenthesized list, in any context where an expression is expected.

3. FILE: See "File Opening and File Attributes" in Chapter 7--in addition, by its appearance in an ON, REVERT, or SIGNAL statement associated with data transmission conditions.

4. TASK: TASK option

5. EVENT: EVENT option or WAIT statement

6. (Programmer named condition): ON CONDITION, SIGNAL CONDITION, or REVERT CONDITION

7. POINTER: CONTROLLED (pointer-variable) declaration, SET option, or pointer qualifier

8. AREA: IN option

Recognition of one of these attributes through context does not redeclare the identifier that is internal to the block in which the contextual reference appeared. If a reference lies within the scope of a declaration (explicit, implicit, or contextual) of the same identifier, the attributes given through the previous declaration and applied defaults must match the attributes given through the contextual reference and applied defaults there. In such a case, the contextually declared identifier is taken to be the same name as that previously declared. Thus, the above contextually determined attributes cannot add to attributes given to the same identifier in a previous declaration.

If an identifier found in one of the above contexts has not been previously declared in a containing block, then a declaration is made for it, internal to the containing external procedure, and the indicated attribute is given. Defaults are then added.

If an identifier appears in a context that furnishes 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 may not add any attributes that are not given explicitly or by default in the DECLARE statement.

For example, the following is illegal:

```
DECLARE F EXTERNAL;
GET FILE(F) LIST(A);
```

Application of Default Attributes

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

ENTRY type: EXTERNAL is assumed. If

the entry is EXTERNAL and is not a subroutine, then REDUCIBLE is assumed. Otherwise, IRREDUCIBLE is assumed. Scale, base, mode and precision defaults for the value returned are the same as for Arithmetic type given below.

If a procedure has multiple entry names and no data attributes, there is potential ambiguity in the characteristics of the value to be returned. In order to avoid this ambiguity, succeeding labels are interpreted as if they were entry names for successive ENTRY statements. For example, in the following, statement a is interpreted as if both statement b and statement c had been written.

a. A: B: ENTRY;

b. A: ENTRY;

c. B: ENTRY;

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

TASK type: ABNORMAL is assumed. Scope and storage class defaults are the same as for Arithmetic type given below. ALIGNED is assumed for arrays not in structures.

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 is assumed. Scope and storage class defaults are the same as for Arithmetic type given below. ALIGNED is assumed for arrays not in structures.

POINTER type: NORMAL is assumed. ALIGNED is assumed for arrays not in structures. Scope and storage class defaults are the same as those for arithmetic type given below.

AREA type: NORMAL is assumed. ALIGNED is assumed for arrays not in structures. Scope and storage class defaults are the same as those for arithmetic type given below.

Condition type: EXTERNAL scope is assumed.

String type: NORMAL is assumed. Scope and storage class defaults are the same as for Arithmetic type given below. ALIGNED is assumed for arrays not in structures.

Major Structure type: PACKED is assumed. NORMAL is assumed. Scope and storage class defaults are the same as for Arithmetic type given below.

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 FLOAT, DECIMAL and REAL. Default precision is implementation defined, dependent on scale and base. ALIGNED is assumed for arrays not in structures. NORMAL is assumed. INTERNAL is assumed. If no storage class is given, then AUTOMATIC is associated with INTERNAL and STATIC with EXTERNAL.

STRUCTURE DECLARATIONS AND ATTRIBUTES

This section is a summarization of data declarations and attributes as they apply specifically to structures.

LEVEL NUMBER

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.

STRUCTURES AND THE DIMENSION ATTRIBUTE

When a structure name is given the dimension attribute, it is an array of structures, and all contained items are arrays (see "Arrays of Structures," in Chapter 2). Contained scalar items, contained structure elements, and cross sections of contained arrays are referred to, respectively, by subscripted names, subscripted qualified names, and the asterisk notation (see "Naming," in Chapter 2).

STRUCTURES AND DATA ATTRIBUTES

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

STRUCTURES AND SCOPE ATTRIBUTES

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.

STRUCTURES AND STORAGE CLASS ATTRIBUTES

All items in the same structure must be 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 form of the CONTROLLED attribute, only the major structure, not its elements, may be allocated and freed.

FORMAL PARAMETERS

The PROCEDURE statement heading a given procedure and defining the primary entry point to the procedure may specify a list of formal parameters. (For syntax and details of the PROCEDURE statement, see Chapter 8.)

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 formal parameters. Formal parameter lists for different entry points to a procedure need not be the same. (For syntax and details see "The ENTRY Statement.")

The formal parameters are identifiers and may appear in statements of the procedure in the context of scalar variable names, array names, structure names, statement label designators, entry names, file names, task names, event names, area names, pointer names, or cell names.

The appearance of an identifier in a formal parameter list for a procedure constitutes a declaration of the identifier as a parameter. This declaration can be combined with an explicit declaration or contextual declarations in the procedure that will associate required attributes with the parameter. Required attributes not declared explicitly or contextually will be assigned by default.

No declarations of the parameter can appear outside the procedure. (For further details about the restrictions on attributes of parameters see "Arguments and Parameters," in Chapter 10.)

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 formal parameter list is (X, Y, Z), or at its secondary entry point SBSEC, where the formal 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 formal 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 (for a definition of "active," see "Activation and Termination of Blocks" in Chapter 6).

When a procedure reference invokes a procedure, each argument specified in the reference is associated with its corresponding formal parameter in the list for the denoted entry point, and control is passed to the procedure at the entry point. The conditions the arguments must satisfy, and the manner of association of each argument with its matching parameter are discussed in "The Arguments in a Procedure Reference."

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. (For a complete description of expression, see "Expressions," in Chapter 3). 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 statement using 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.

(Ordinarily a given procedure will be used exclusively as a function procedure or exclusively as a subroutine procedure.)

FUNCTION REFERENCES AND FUNCTION PROCEDURES

When a function reference appears in an expression, the function 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, in place of the function reference, 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, or pointer type (see "The RETURN Statement"). (A GO TO statement may also be used to terminate execution of a procedure invoked by a function reference.) 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 (see "The PROCEDURE Statement" and "The ENTRY Statement," in Chapter 8). 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.

If the invoked function procedure is terminated by a GO TO statement, the evaluation of the expression that invoked the function will not be completed and control will go to the designated statement.

GENERIC FUNCTIONS

A generic function is a family of functions with a single name. A function reference to a generic function 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 functions may be built-in (see below) or specified by the programmer, who may, by means of the attribute GENERIC, define a name to be a generic function name. An entry name may be explicitly declared with the GENERIC attribute. The GENERIC attribute requires a list of all of the entry names of the family and the attributes of all of the arguments for each member (different members must have different argument attribute patterns). 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 that has the same argument attribute pattern as the pattern in the argument list of the reference. For complete details see "Entry Name Attributes" in Chapter 4.

Subroutine procedures may also be generic. The method of selecting a particular subroutine corresponds exactly to that of selecting a particular function.

BUILT-IN FUNCTIONS

Besides function 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 complete list of these functions and their descriptions can be found in Appendix 1.

A large number of the built-in functions are generic. The built-in generic functions are of considerable convenience to the programmer. He may, for example, always use the same name EXP for the exponential function, regardless of whether the argument is of REAL or COMPLEX mode, regardless of the precision of the argument, etc., and automatically he will obtain that one of the EXP family that fits the requirements.

Each built-in function, whether or not it is generic, has a specified number of arguments given. For some built-in functions only a minimum is specified; additional arguments are optional. For others, a maximum is specified; only one argument is required.

Each of the built-in functions that are not generic has only a single member. When a reference is made to one of these func-

tions, 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.

The fixed set of names for the built-in functions is part of the language of PL/I. However, the identifiers corresponding to these names are not reserved; any such identifier can be used by the programmer for other purposes. If the identifier is declared explicitly for some other use, any appearance of the identifier in the scope of this declaration will refer to that other use. The built-in function cannot, of course, be used in this scope. If the identifier appears, but not in the scope of a declaration establishing the identifier for another use, the identifier will be regarded as implicitly declared in the containing external procedure with the attribute BUILTIN, and this appearance will refer to the built-in function.

If an identifier corresponding to a built-in function name is declared to have a use other than as the built-in function in some block, the built-in function can be used in contained blocks by declaring the identifier with the attribute BUILTIN.

SUBROUTINE REFERENCES AND SUBROUTINE PROCEDURES

When a procedure is invoked as a subroutine by the execution of a CALL statement or a statement with 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 formal parameters and control is passed to the procedure at the denoted entry point. (If the invocation involves a task option, the procedure will not necessarily be activated immediately; see "Asynchronous Operations and Tasks" in Chapter 6.)

Unlike the function procedure, the subroutine procedure does not return an explicitly specified value to the point of invocation. The procedure may terminate in the following ways:

1. Control reaches a RETURN statement for the procedure. When executed, this statement returns control to the first executable statement logically following the invoking statement, unless the invocation specified a task option or the procedure was invoked by a statement with a CALL option. If a task option has been used, control is simply terminated for this task. If the procedure was invoked by a statement having a CALL option, control is returned to that statement at the point immediately following the CALL option.

2. Control reaches an END statement for the procedure, which in this case is treated as a RETURN statement. The effect is as in case 1.

3. Control reaches a GO TO statement in the procedure that transfers control out of the procedure. (This is not permitted if the procedure has been invoked by a statement with a CALL option or in a CALL statement with a task option.) In this case, control will go to the designated statement (see "The GO TO Statement"). The statement label designator of the GO TO statement may be a parameter of type LABEL, which is associated with a label argument passed from the invoking procedure.

4. Control reaches an EXIT or STOP statement.

Example of Function Reference:

```
COMP:  PROCEDURE;
          .
          .
          .
       S1:  P10=Q5*POLY5(R0, VAL1);
          .
          .
          .
       POLY5:  PROCEDURE (C, X);
               RETURN(C+X*(1+X*(2+X*(3+X*(4
                 +5*X)))));
               END POLY5;
          .
          .
          .
       END COMP;
```

In this example, the external procedure COMP contains the function procedure POLY5, which is invoked when the expression Q5*POLY5(R0, VAL1) is being evaluated during execution of the assignment statement labeled S1. When POLY5 is invoked, the arguments R0 and VAL1 will be associated with the parameters C and X, respectively.

The returned value for POLY5 (R0, VAL1) will be the value of the expression:

```
R0+VAL1*(1+VAL1*(2+VAL1*(3+VAL1*(4+5*
   VAL1))))
```

Examples of Subroutine Reference:

```
1. COMP:  PROCEDURE;
                .
                .
                .
          S1:  CALL POLY5 (R0, VAL1);
          S2:  P10 = Q5*TEMP;
                .
                .
                .
                .
                .
          POLY5:  PROCEDURE (C, X);
                  TEMP=C+X*(1+X*(2+X*(3+X*
                  (4+5*X))));
                  RETURN;
                  END POLY5;
                .
                .
                .
          END COMP;
```

In the above example, the effect is the same as in the previous example using the function reference. The subroutine procedure POLY5 is invoked by the CALL statement labeled S1. The arguments and parameters are associated as in the previous example, but here, the value of the expression (the same as in the previous example) is assigned within the subroutine to the variable TEMP, which is used by the statement labeled S2, after the RETURN statement passes control back to that statement. Thus, communication of the value is by means of the shared variable TEMP, which, of course, remains available for use following the execution of S2.

In some cases the invoked and the invoking procedure may be separated in such a way that sharing a name in the above simple manner is not possible (see "Scope of Declarations"). Another more general method of communicating values from the invoked procedure, which may be applied in these cases, is illustrated in the following alternative example:

```
2. COMP:  PROCEDURE;
                .
                .
                .
          S1:  CALL POLY5 (R0, VAL1, TEMP);
          S2:  P10=Q5*TEMP;
                .
                .
                .
          POLY5:  PROCEDURE (C,X,Z);
                  Z=C+X*(1+X*(2+X*(3+X*
                    (4+5*X))));
                  RETURN;
                  END POLY5;
                .
                .
                .
          END COMP;
```

Here, the invocation of POLY5 by the CALL statement will associate the variable TEMP with the parameter Z, and the action will be exactly as in the previous example: the parameter Z will effectively be replaced by the name TEMP in the assignment statement for Z, and TEMP will be assigned the value of the expression on the right-hand side, with R0 replacing C and VAL1 replacing X, before return to statement S2. In this case, the value has been communicated from the subroutine through a parameter.

The above two examples illustrate how a single value obtained in a subroutine can be communicated back to the invoking procedure. The action of a subroutine will generally be more complex than this; many communicated variables may be involved, whether scalar, array, structure, or statement-label variables; input/output operations may be specified, etc. In contrast, the usual purpose of a function procedure is to return a scalar value.

THE ARGUMENTS IN A PROCEDURE REFERENCE

In general, the arguments in a procedure reference may be any of the following:

1. Expressions
2. Data elements
3. Entry names (programmer-defined)
4. Built-in Float Arithmetic Generic Function names (see Appendix 1)
5. 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. (An exception in case of string arithmetic data arguments is described below.)

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 formal 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 have 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.

If an argument is an entry name with no argument list, the entry name (rather than the function value) is always passed, independent of whether the entry name requires parameters.

Example:

```
DECLARE RANDOM ENTRY RETURNS
(FLOAT);
L1: CALL SUB(RANDOM);
L2: CALL SUB1(Y*RANDOM);
```

In statement L1, the entry name RANDOM is passed. However, in statement L2, the value of the function RANDOM is required, and this value, multiplied by Y, is passed.

Note: This rule also applies for arguments to built-in functions.

THE USE OF THE ENTRY ATTRIBUTE

An identifier is contextually declared to be an entry name in a block if it

1. appears as a label to a PROCEDURE or ENTRY statement or
2. appears in the block following the keyword CALL or
3. appears as the function name in a function reference that contains an argument list.

If it is desired to use the identifier as an entry name in a block where it is not so declared, the identifier must be given the ENTRY attribute explicitly in a DECLARE statement for the block.

As an illustration, in the above example, the CALL statement:

```
CALL SUB(R*S, CALC, L5);
```

has the entry name CALC as its second argument. This appearance of CALC is not recognizable as an entry name by context. It must previously have been declared (either contextually, or explicitly in a DECLARE statement) to have the attribute ENTRY.

A more general form of the ENTRY attribute allows the programmer to enumerate the attributes of the parameters for the named entry point.

As an illustration, in the above CALL statement example, the three parameters corresponding to the three arguments of the CALL statement might be described in the invoking procedure by the statement:

```
DECLARE SUB ENTRY (FIXED, ENTRY,
   LABEL);
```

This statement specifies that:

1. SUB is an entry name.
2. The entry point SUB has three parameters.
3. The first parameter has the FIXED attribute with certain default data attributes.
4. The second parameter has the ENTRY attribute.
5. The third parameter has the LABEL attribute.

The number of parameters and the attributes of each, as described in the ENTRY attribute specification, must always agree with the number of parameters and their attributes, as defined for the described entry point within the invoked procedure.

One of the applications of the extended form of the ENTRY attribute is mentioned in the immediately following description. (A detailed discussion of the various uses for the ENTRY attribute, including the IRREDUCIBLE, USES, SETS, and GENERIC attributes, can be found in Chapter 4.)

PASSING ARGUMENTS TO THE ENTRY POINT

When a procedure is invoked at a given entry point by a procedure reference and each argument is associated with its corresponding formal parameter, the arguments are said to be passed to the entry point.

The action involved in passing the arguments generally will assume that the attributes of each argument match the attributes of its corresponding formal parameter, as described above. However, if the argument is an expression whose attributes do not correspond to those declared for the parameter associated with that argument, the expression will be evaluated and converted, before the argument is passed, to conform to the attributes described by the corresponding member of the ENTRY attribute list.

As an illustration, in the preceding example, the first argument in the CALL statement, which invokes the procedure SUB, is the expression R*S. Assume that R*S has the FLOAT attribute with certain default attributes. These do not match the attributes of the first parameter at the entry point SUB. Then the ENTRY attribute must be used in the invoking procedure to specify the same attributes for the first parameter as specified in the invoked procedure SUB. (The preceding illustration shows one way of doing this.) Thus, on execution of the CALL statement, the expression R*S is evaluated, giving a floating-point result, which is then converted to a fixed-point value with the other required attributes, before being passed to the entry point SUB.

(A detailed description of the action involved in passing arguments to the invoked entry point can be found in Chapter 10.)

In certain circumstances, the preparatory action includes the construction of a dummy argument. For example, a dummy argument is constructed when the argument must be converted, as in the example of R*S just discussed, or when the argument is an expression involving constants or operators (R*S is again an example of this circumstance).

In each of its appearances as a reference in the procedure, the formal parameter corresponding to the argument effectively is replaced by the argument name. Thus, all appearances of the parameter during execution of the procedure are treated as appearances of the argument name. However, in the cases where a dummy argument is constructed, it is the dummy argument name that replaces the parameter. Passing an argument does not always imply a true logical substitution of the argument name for the parameter in the procedure. However, in the important case where the argument is an arithmetic, string, or label variable having identical attributes with the corresponding parameter, a logical substitution does occur. Thus, parameters can be used to communicate values from the invoked procedure back to the invoking procedure. Example 2 of "Subroutine References," above, is an illustration of this.

In the above example, the appearance of CALC as the second argument when SUB is called does not imply that the identifier CALC is contextually declared as an entry name, even though the above ENTRY attribute for SUB has been given.

THE SPECIAL PROCEDURE ATTRIBUTE RECURSIVE

In the PROCEDURE statement for a given procedure, certain special attributes that characterize the procedure itself may be specified. (For a complete discussion of these attributes, see "The PROCEDURE Statement.") One of these, which has particular significance, is the attribute 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 attribute. (See "Data Known to Invocations of Recursive Procedures" in Chapter 10 for additional details.)

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, which invokes the initial procedure. This initial procedure must be an external procedure that may be specified with an attribute in the options list of the OPTIONS attribute (see "The PROCEDURE Statement" in Chapter 8). This procedure cannot have CONTROLLED parameters (see "Storage Classes" in this chapter).

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.

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.

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; see "Asynchronous Operations and Tasks.")

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 immediate dynamic descendant 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 dynamic descendant of B.

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

Block A dynamically encompasses block B, or block B is dynamically encompassed by block A, if B is a dynamic descendant of A.

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. On the other hand, the entire problem of allocation can be ignored completely by the programmer, if storage economization is of little significance in his situation, and a reasonably efficient use of storage usually will still be obtained automatically.

DEFINITIONS AND RULES

Storage is said to be allocated for a variable when a certain region of 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 it 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).

At any point where a variable appears as a reference, it must be in an allocated state.

Note: An unallocated variable may appear as an argument to a procedure with a corresponding CONTROLLED parameter, as an argument to the ALLOCATION function, or in an ALLOCATE statement.

STORAGE CLASSES

Every variable in a program must have a storage class, which specifies the manner of storage allocation.

There are three storage classes. The storage class is specified by declaring the variable with one of the three storage class attributes STATIC, AUTOMATIC, or CONTROLLED (based or nonbased). The storage class may be declared explicitly or by default.

The Static Storage Class

Storage for a variable with attribute STATIC is allocated before execution of the program and is never released during execution.

The scope attribute (see Chapter 4) 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 status of the block containing this variable (see "Data Description") determines dynamic 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 the block. At the time of termination, the storage will be released.

Thus, the time interval during which the variable is in an allocated state will necessarily include the intervals when the variable is known (see "Scope of Declarations").

Termination of a block by means of a GO TO statement may imply simultaneous termination of other blocks and, consequently, simultaneous release of storage for all AUTOMATIC variables declared in these blocks (see "The GO TO Statement").

If the block is a procedure and is called recursively (reactivated one or more times before return), previously allocated storage for the AUTOMATIC variable 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 (see Chapter 4) of an AUTOMATIC variable must be INTERNAL. An INTERNAL variable with unspecified storage class has, by default, AUTOMATIC storage class attribute.

The Controlled Storage Class

The ALLOCATE statement (see Chapter 8) specifies one or more variables, each with certain optional attributes. Execution of the statement causes the allocation of storage for the variable specified.

The following four paragraphs apply only to nonbased controlled variables.

If a variable has the attribute CONTROLLED, storage allocation must be explicitly specified for the variable by the ALLOCATE and FREE statements.

The FREE statement specifies one or more 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 variable X is in an allocated state. The built-in function ALLOCATION (see Appendix 1) is provided to test this state. The function reference ALLOCATION (X) will return the value '1'B if X is in an allocated state, and the value '0'B if not.

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 CHAR-
            ACTER (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 entrance 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 allocation of based variables is discussed in "The ALLOCATE Statement" (Chapter 8) and in "List Processing" (Chapter 10).

ASYNCHRONOUS OPERATIONS AND TASKS

PL/I allows tasks to be created by the programmer and provides facilities for the following:

1. Synchronizing tasks
2. Testing whether or not a task is complete
3. Changing the priority of tasks

SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS

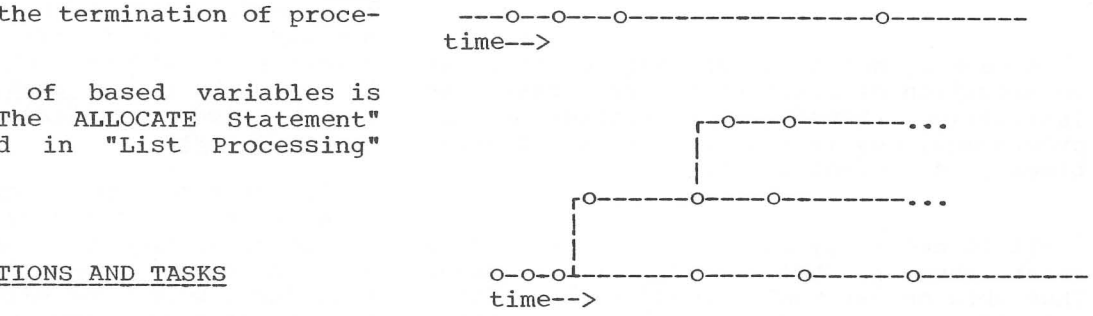
Unless the program specifies the creation of tasks, the execution of the statements of the program will proceed serially in time, according to the sequence designated by the order of the statements and the control statements (see "Sequence of Control" in Chapter 8). Such operation is said to be synchronous.

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.)

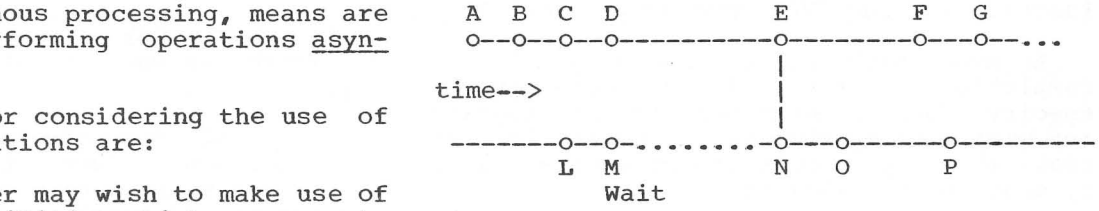


In asynchronous operation, once a new line has been started, the statements on that line are executed in sequence, but independently of the statements on any other line. Statements on any two lines need not necessarily be executed simultaneously -- whether this occurs depends on the resources and state of the system.

SYNCHRONIZING TWO ASYNCHRONOUS OPERATIONS

In order that the result of an asynchronous operation may be made available to other procedures, means are provided to synchronize two or more asynchronous operations.

The following diagram illustrates this:



Assume that before statement N can be executed, both M and E must have been executed. M therefore issues a WAIT statement which will suspend operation on that line until E has completed. After N, the statements O, P,..., are executed synchronously, as are the statements F, G,...,.

TASK AND EVENTS

In PL/I, asynchronous operations result from the creation, by the programmer, of tasks. 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, the programmer has to create new tasks, as described below. All tasks created by the programmer are called subtasks.

With each task, except the major task, it is possible to associate a task name. The task name may be used to refer to and set the priority 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 name. (See the EVENT built-in function and the EVENT pseudo-variable.)

An event name may be associated with the completion of a task. It is necessary to specify such an event name if the programmer wishes to synchronize a point in one task with the completion of another task, by means of the WAIT statement.

Other event names may be defined by the programmer and used in WAIT statements. In this way, the programmer can synchronize a task with events other than the completion of another task. Event names may be set by referring to them in assignment statement by means of the EVENT pseudo-variable.

THE CREATION OF TASKS

In PL/I tasks are created by writing:

A TASK option
An EVENT option
A PRIORITY option

or any combination of these options in a CALL statement (see "The CALL Statement" in Chapter 8). 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.

The task created by the CALL statement must be given a priority. This priority may be given in either of two ways:

1. through the PRIORITY option in the CALL statement, or
2. by assignment to the PRIORITY pseudo-variable prior to the execution of the CALL statement that creates the 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 (i.e., completed) in one of the four following ways:

1. Control for the task reaches an EXIT statement (see Chapter 8 for a discussion of each of the statements mentioned here).
2. Control for any task reaches a STOP statement.
3. Control for the task reaches a RETURN statement for the procedure invoked with a task option.

4. Control for the task reaches an END statement for the procedure invoked with a task option.

ALLOCATION OF DATA IN TASKS

The rules of scope and storage allocation hold across task boundaries. If storage is allocated for a variable in the attaching task, this allocation may apply to the attached task, so that the variable may appear as a reference in the attached task. It is the responsibility of the programmer to be certain that storage for such a variable is not released too early in the attaching task. (Normally, this is done by synchronizing by use of the WAIT statement.)

(Further details concerning tasks as related to storage allocation and other special considerations can be found in Chapter 10; also see "The WAIT Statement" for additional information and examples.)

INTERRUPT OPERATIONS

During the course of program execution any one of a certain set of conditions may occur that can result in an interrupt. 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. The time point of an interrupt is, in general, unpredictable.

For most conditions that can cause an interrupt, the special action to be taken 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 ON-conditions. A complete list and description of the ON-conditions can be found in Appendix 3. With two exceptions (see "Programmer Defined ON-Conditions," in this chapter), each ON-condition is named with a unique identifier suggestive of the condition (e.g., ZERODIVIDE names the condition obtaining whenever an attempt is made to divide by zero). This collection of names, like the built-in function 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 usually unpredictable for a statement in which a disabled condition occurs. However, in certain cases the results are defined (e.g., the CHECK condition).

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

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 scope 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 re-specified. The scope does not include procedures that lie outside the scope as defined above but which may be invoked by the execution of statements in this scope.

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.

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, which has the general form:

```
ON condition-specification action-specification
```

The "condition specification" either is an ON-condition name or denotes a programmer-defined condition, and the "action specification" is a single simple statement or begin block, optionally preceded by the keyword SNAP (see "The ON Statement" for complete syntax and details). If the single statement is null, control is given back to the point of interrupt.

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 activity 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 action specification 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 action specification 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 eight ON-conditions whose names (possibly preceded by the word "NO") may appear in a prefix to a statement. 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 three of these eight ON-conditions, SIZE, SUBSCRIPTRANGE, and CHECK (identifier list), for which an interrupt will not take place when the condition occurs unless the programmer specifically designates that the interrupt <u>is</u> 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 eight special ON-conditions, namely OVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, and FIXEDOVERFLOW, are always enabled, but the programmer may specifically designate that an interrupt is <u>not</u> 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.

All other conditions, whose names cannot be used in a prefix, are always enabled.

SYSTEM INTERRUPT ACTION

Each of the ON-conditions has a standard action defined for it if an interrupt should occur. If there has been no pre-

vious execution of an ON statement (in which the programmer specifies the interrupt action), any interrupt caused by the occurrence of the condition during program execution will result in a standard system action, which 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 special action-specification 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 error (see Appendix 3) 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 error 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 error 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 errors 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:

```
(NOOVERFLOW):  A:  PROCEDURE;
                     .
                     .
                     .
    (OVERFLOW):B:  BEGIN;
                     .
                     .
                     .
                   END B;
                     .
                     .
                     .
                   END A;
```

In the above example, interrupts will be suppressed for OVERFLOW conditions occurring during execution of that part of procedure A that is not included in block B. OVERFLOW conditions occurring during execution of block B will result in an interrupt.

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 without having to return control to that block (see "The REVERT Statement," in Chapter 8 for format and rules).

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 error 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 error that occurs in procedure B will cause an interrupt with the action GO TO AERR.

PROGRAMMER-DEFINED ON-CONDITIONS

There are two kinds of ON-conditions the programmer may construct:

1. An arbitrary 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) action-specification

   Such a statement contextually declares the "identifier" to be a condition-name and the execution of the statement provides an action specification. 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(KEY) block
   ```

   and later the following statement is executed:

   ```
   SIGNAL CONDITION(KEY);
   ```

   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.

2. The CHECK (identifier list), where "identifier list" represents variables or labels declared in the program, can appear as the condition specification in the ON statement. Whenever one of the variables in the list is assigned a value, or one of the procedures or statements whose label appears in the list is executed and if the condition is enabled, the condition defined by this specification is regarded as occurring, and an interrupt will take place. (For a precise explanation of this kind of condition, see Appendix 3, "ON Conditions.")

FACILITIES FOR PROGRAM CHECKOUT

The programmer-specified condition described above is a powerful tool for program checkout. As an example of its use, suppose that a block contains the prefix (CHECK(A,SUB1,ST5)) and that the following statement is executed:

```
ON CHECK (A, SUB1, ST5) SYSTEM;
```

In the example, A is a data variable, SUB1 is a procedure name, and ST5 is a statement label. Then, whenever a value is assigned to A (or to any part of A, if A is an array or structure name), an interrupt occurs, and A is printed out on the standard output file (SYSPRINT) with its new value. If the statement labeled ST5 or the procedure SUB1 is executed, the label is printed out.

Another useful ON-condition is the condition named SUBSCRIPTRANGE. Parts of the program can be designated by the programmer, using the keyword SUBSCRIPTRANGE in appropriate prefixes, to receive constant monitoring of subscript values. Whenever the value of some subscript in some array goes out of its designated range, an interrupt will occur, and action, specified by a previously executed ON statement, may take place to correct the error.

The SIGNAL statement also will be found useful for checkout, since it can be used to simulate the occurrence of any ON-condition (see "The SIGNAL Statement").

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 is regarded as 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, addressable buffer. When transmission is to or from data variables, the attributes of the variables should accurately describe the composition of the record.

For annotated illustrations of input/output operations, see Examples 1 and 2 in Appendix 6.

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 never are applied by default and must always be stated explicitly, 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 (decimal-integer-constant)
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 specified through the TITLE option of the OPEN statement; otherwise, the filename will specify 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.

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. 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 deduced from each:

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--explicitly or implicitly--as the result of the file opening. 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;
```

Attributes after merge due to execution of the OPEN statement are STREAM and PRINT.

Attributes after implication are STREAM, PRINT, and OUTPUT.

Attributes after default application are STREAM, PRINT, OUTPUT, and EXTERNAL.

Implicit opening example

```
DECLARE   MASTER FILE KEYED (10)
                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 (10), INTERNAL, RECORD, and INPUT.

Attributes after implication are KEYED (10), INTERNAL, RECORD and INPUT. There are no additional attributes implied.

Attributes after default application are KEYED (10), INTERNAL, RECORD, INPUT, SEQUENTIAL, and BUFFERED.

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 data specifications correspond to the modes of transmission.

DATA LISTS

List-directed, data-directed, and edit-directed data specifications require a data list to specify the data items to be transmitted.

General format:

(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, a pseudo-array, a pseudo-structure, 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.

Repetitive Specification

General format is shown in Figure 1.

```
(element [, element]... DO  { variable        } = specification
                            { pseudo-variable } [, specification]...)

A specification has the following format:

                [ TO expression-2   [BY expression-3] ]
expression-1    [                                     ]   [WHILE (expression-4)]
                [ BY expression-3   [TO expression-2] ]
```

Figure 1. General Format for Repetitive Specification.

Syntax rules:

1. Each element in the element list of the repetitive specification is described for data-list elements above.

2. 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.

3. Repetitive specification may be nested to a depth whose maximum is implementation-defined. 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 the elements of the array A in the following order:

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

Note: The DO keyword is used in the repetitive specification to indicate iteration in a manner similar to a DO statement. A corresponding END statement is not required.

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 1A(10), 2B, 2C;
```

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 1A, 2B(10), 2C(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, 1B, 2P, 2E, 3F,
    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 may not be used. A string constant must be one of the two permitted forms listed above. Iteration and string repetition factors are not allowed.

Redundant blanks are permitted as in PL/I programs. However, no blanks may precede the central + or - in complex expressions.

Data items in the stream must be separated either by a blank or by a comma. This separator may be surrounded 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 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 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.

String Value	List item	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.

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.

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.

A blank is used to separate data items transmitted.

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 significant digits in the data field.
4. The letter q represents the number of digits to the right of the decimal point.
5. The letter s represents a scale factor as described for floating-point data.
6. The letters yyy represent a scale 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 symbol b represents a blank position in the output.

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.

Coded Real Fixed-Point Decimal Data: The data item is converted to precision (p,q), plus a possible scaling field. It is transmitted to a field of width w, plus the scaling field if it is present.

If q is greater than or equal to zero and less than or equal to p, then w = p+3, and d=q; for example:

```
bbxxxx.xxxx        (p=8,q=4)
bbbxxxxxxxx        (p=8,q=0)
```

If q is less than zero or greater than p, then w = p+3+n, where n is the number of digits required to express q; for example:

```
bxxxxxxxxF-yyy     (p=8, q=100, yyy=-q)
```

Zero suppression is performed to the left of the field, and if the value is less than zero, a minus sign will immediately precede the first significant digit.

Coded Real Fixed-Point Binary Data: The data item is converted to fixed-point decimal and is transmitted as coded real fixed-point decimal data.

Coded Real Floating-Point Decimal Data: The data item is converted according to the rules for floating-point format items, E(w, d, s). For E-conversion, w = p + 6, d = p - 1 and s = p.

Coded Real Floating-Point Binary Data: The data item is converted to floating-point decimal with a precision (p) and transmitted as coded real floating-point decimal data.

Coded Complex Data: 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.

NUMERIC FIELD DATA: The base of numeric field data is either decimal or binary.

Numeric Decimal Data: The external format and field width of the numeric decimal data item is that described by the associated picture specification.

Numeric Binary Data: 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.

Complex Numeric Data: The real and imaginary parts are output as above and the external representation is the concatenation of the real and imaginary parts. The field width is 2w, where w is the number of bytes (or bits, if binary) allocated to the real part of the numeric data; no I is appended.

CHARACTER-STRING DATA: 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 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 DIS-
   TANCE = 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 may not include formal parameters, based or defined variables. Names of structure elements need only have enough qualification to resolve any ambiguity; full qualification is not required.

2. Option 1 implies that all of the data items to be transmitted are known to the block containing the GET statement. Option 1 may be used for data-directed input only.

3. Option 2 may be used for both data-directed input and output.

4. 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 for list-directed transmission.

General rules for data specifications of 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.

   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 identical form 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;

General rules for data-directed output:

1. The elements of the data list may be a scalar name, an array name, a structure name, 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.

2. Array names 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. Subscript expressions in a data name

are evaluated and replaced by integer constants.

4. 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;

5. 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:

   1A, 2B, 2C, 3D

   If a data list for data-directed output is as follows:

   DATA (A)

   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;

6. When p<q or q<0, data-directed output of FIXED data of precision (p,q) is not suitable for data-directed input.

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").

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. If it is desired to calculate values, 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 is described in "Data Lists," the format list in "Format Lists." This form of transmission can be used for sterling data.

```
AB:  PROCEDURE;

     DECLARE A(6), B(7);

     GET FILE (X) DATA (B);

     DO I = 1 TO 6;

     A (I) = B (I+1) + B (I);

     END;

     PUT FILE (Y) DATA (A);

     END AB;
```

Input Stream

```
B(1)=1, B(2)=2, B(3)=3,
B(4)=1, B(5)=2, B(6)=3, B(7)=4;
```

Output Stream

```
A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
A(5)= 5 A(6)= 7;
```

Figure 3. Example of Data-Directed Transmission, both Input and Output

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, the second scalar data item with the second format item, etc. Suppose the format list effectively contains j format items, and the data list effectively contains k data items. Then, if j<k after j 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. 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.

4. If a data list item is associated with a control format item, that control action is executed and the data list item is paired with the next format item.

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:

```
( { 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. 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.

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 scale factor, which may be positive or negative.

The quantities w, d, s, and p may be specified by an expression. When the format item is used, the expression is evaluated and converted to an integer. If w≤0 in a format specification, then, on input, the associated data and format list items are skipped, unless it 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 converted to the characteristics of the list item. Rules for the conversion are given in Chapter 3.

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.

   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 scale 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.

   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).

   In Option 3, the scale 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 number 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.

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.

   The external form of the number is as follows:

   [±] fixed-point-number [{[E] ±} exponent]
                          { E [±] }

   (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 fixed-point number. 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. 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 inte-

ger of n digits, where n is defined individually for each implementation. The exponent is adjusted so that the leading digit of the fractional part is nonzero.

(b) If the above form does not fill the field of width w, it is right-adjusted, and blanks are inserted on the left. 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.

COMPLEX FORMAT ITEMS: Complex numeric 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 or character field.

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 enclosed 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. The external representation of binary numeric fields uses the characters 0 and 1.

On output, the value of the list item is edited to the form specified by the picture before it is transmitted. Binary numeric fields will have a character representation after transmission.

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.

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 "character picture specification" is described in "The String Attributes", in Chapter 4.

2. The external representation is a string of w characters.

3. 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.

4. On output, w can be omitted for string list items, in which case w is taken to be the current length of that string. On input, w is always required.

Control Format Items

There are two types of control format items, the spacing format item X and the printing format items.

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.

Printing Format Items

The printing format items can be used only with STREAM PRINT files. There are four of them.

General format:

```
PAGE
SKIP [(w)]
LINE (w)
COLUMN (w)
```

General rules:

1. The PAGE, SKIP, and LINE format items operate in the same manner as the corresponding options with the PUT statement.
2. The COLUMN (w) format item specifies that blank characters are to be inserted into the stream so that the next character will be the wth character of the current line. If at least w characters already have been written on the current line, the current line is completed, a new line is started, and w-1 blanks are inserted in it so that the new current line begins at the wth character. If w is greater than LINESIZE as specified in the OPEN statement, or is less than 1, then w is assumed to be 1.

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 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 designator, 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.

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; the OPEN and CLOSE statements, which may also be used in STREAM transmission, are discussed earlier in this chapter).

STREAM INPUT:

```
GET [FILE (filename)                          ]
    [STRING (character-string-variable)       ]
            data-specification [COPY];
```

STREAM OUTPUT:

```
PUT [FILE (filename)                          ]
    [STRING (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 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	which may be used with any STREAM file
PAGE, SKIP [(w)], LINE (w), COLUMN (w)	which may be used only with STREAM OUTPUT PRINT files

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 the storage class, AUTOMATIC, STATIC or CONTROLLED. The variables may not be formal parameters or defined variables. In addition, they must not contain VARYING length strings. They may contain LABEL and POINTER variables, but such data may lose its validity in transmission.

With RECORD transmission, it is possible to operate upon the record in a buffer if the file has the BUFFERED attribute. Operation within the buffer can be 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. A based variable and its associated pointer variable are specified by the following form of the CONTROLLED storage class attribute specification:

```
CONTROLLED (pointer-variable)
```

The pointer variable, itself, may have any storage class attribute; however, the default is AUTOMATIC. The pointer variable also may be given either INTERNAL or EXTERNAL scope attribute, with default being INTERNAL; but the scope of the based variable is INTERNAL. The EXTERNAL attribute cannot be specified.

Consider the following declaration:

```
DECLARE  1 MASTER_RECORD CONTROLLED
           (REC_IDENT),
             2  IDENTIFICATION
                CHARACTER (10),
             2  NAME CHARACTER
                (30),
             2  ADDRESS,
                3  STREET
                   CHARACTER
                     (15),
                3  CITY
                   CHARACTER
                     (15),
                3  STATE
                   CHARACTER
                     (15),
                3  ZIP
                   CHARACTER
                     (5);
```

The name MASTER_RECORD is the based variable which can be used to describe a record in the buffer that conforms to the attributes declared for MASTER_RECORD. REC_IDENT is a pointer variable that identifies the position of MASTER_RECORD within the buffer. The pointer variable has the default storage attribute of AUTOMATIC. The based variable, of course, is explicit-

ly declared to have the CONTROLLED storage class attribute.

If any attributes other than AUTOMATIC are to be declared for a pointer variable, they must be explicitly declared. For example, the following declaration specifies the STATIC and EXTERNAL attributes for the pointer variable REC_IDENT:

```
DECLARE REC_IDENT POINTER STATIC
                  EXTERNAL;
```

Note: In this declaration, the POINTER attribute is declared explicitly. In the previous example, the POINTER attribute was declared contextually by the appearance of the pointer variable name in the CONTROLLED attribute specification.

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; the OPEN and CLOSE statements, which also may be used in RECORD transmission, are discussed earlier in this chapter). A general discussion of RECORD transmission follows this summary.

SEQUENTIAL BUFFERED INPUT:

```
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);
```

SEQUENTIAL BUFFERED OUTPUT:

```
WRITE FILE (filename)
    FROM (variable)
    [KEYFROM (expression)];

LOCATE  variable  FILE (filename)
    SET (pointer-variable)
    [KEYFROM (expression)];
```

SEQUENTIAL BUFFERED UPDATE:

```
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);
```

SEQUENTIAL UNBUFFERED INPUT:

```
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)];
```

SEQUENTIAL UNBUFFERED OUTPUT:

```
WRITE FILE (filename)
    FROM (variable)
    [KEYFROM (expression)]
    [EVENT (event-variable)];
```

SEQUENTIAL UNBUFFERED UPDATE:

```
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)];
```

DIRECT INPUT:

```
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 (expressions);
```

RECORD TRANSMISSION OPERATIONS

The following points cover the salient environmental factors in the use of RECORD transmission:

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.

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 be KEYED. If it is KEYED, the keys may be ignored while accessing sequentially, or they may be extracted from the data set or placed into the data set by the KEYTO and KEYFROM options. 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, modified, or ignored, but the number of records cannot be increased or decreased. Operation with a DIRECT UPDATE file, however, may specify that records are to be added to the data set, through use of the WRITE statement, or deleted from the data set, through use of the DELETE statement. An existing record in an UPDATE file can be replaced through use of a REWRITE statement.

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 that are read from the data set. 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 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, UNLOCK, or CLOSE statement. Records are unlocked automatically 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 filename; if they refer to the same data set using different filenames, the protection does not apply. In addition, the data set to which reference is made by more than one task through the same filename must be opened by the parent of all these tasks.

10. A WRITE statement adds records to a data set, while a REWRITE statement replaces records. Thus, a WRITE statement may only be used with OUTPUT files, and a REWRITE statement may only be used with UPDATE files. Moreover, a WRITE statement uses the KEYFROM option to indicate the actual transference of the key from internal storage to the data set; the REWRITE statement uses the KEY option to identify the existent record to be replaced.

STANDARD FILES

There are two standard system files that are available for use by a PL/I program. The first is a standard system input file called SYSIN. The second is a standard system print file called SYSPRINT. The keywords GET and PUT without a file or string name are equivalent to:

```
GET FILE(SYSIN)...;
PUT FILE(SYSPRINT)...;
```

The implicit reference to the standard files applies only in the GET and PUT statements. Any other reference to either file must be stated explicitly.

The standard files may be given other file attributes explicitly or contextually, but unless SYSPRINT is explicitly declared by the programmer to have the INTERNAL scope attribute, the PRINT attribute is applied automatically.

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, data 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 modified by the following statements: CALL, END, EXIT, GO TO, PROCEDURE, RETURN, SIGNAL, and STOP.

A CALL statement passes control to the specified entry point.

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.

PSEUDO-VARIABLES

The following built-in functions (see Appendix 1 for a more complete description) may be used as pseudo-variables on the left side of an equal sign in an assignment statement, or a DO statement, or in a data list in a GET statement. In the definitions below, the item in the data list of a GET statement may be considered to correspond to the item on the left side of the equal sign in an assignment statement; the value being transmitted may be considered to correspond to the expression on the right side.

COMPLEX (a,b) The letters a and b represent variables that need not have the same characteristics. During execution of an assignment statement, the real part of the expression on the right is assigned to a, the imaginary part to b.

REAL (c) The letter c represents a complex variable. During execution of an assignment statement, the real value of the expression is assigned to the real part of c.

IMAG (c) The letter c represents a complex variable. During execution of an assignment statement, the real value of the expression is assigned to the imaginary part of c.

ONSOURCE (Used in the on-unit of an ON CONVERSION statement) The expression on the right of the equal sign is evaluated, converted to a character string, and assigned to the string that caused the conversion error. The string will be padded with blanks, if necessary, to the length of the string that caused the error.

ONCHAR (Used in the on-unit of an ON CONVERSION statement) The expression on the right of the equal sign is evaluated, converted to a character string of length one, and assigned to the character that caused the error.

SUBSTR (s,i[,k]) The letter s represents a string. During execution of an assignment statement, the expression is assigned to the substring of s defined by the built-in function SUBSTR (see Appendix 1). This substring is always treated as a fixed length string.

EVENT(v) The letter v represents a scalar or array event name. When used in an assignment statement, the expression on the right-hand side is evaluated and converted to a bit string of length 1. The value of this bit string is used in an assignment to the named event (see "Asynchronous Operations and Tasks" in Chapter 6).

PRIORITY[(v)] The letter v represents a scalar or array task name. When used in an assignment statement, the expression on the right-hand side is evaluated and converted to FIXED (m,o) where m is implementation-defined. The priority of v, the named task, is adjusted to be n, relative to that of the task in which the assignment is performed, prior to that assignment. If v is not specified, this is the task in which the assignment statement is executed (see "Asynchronous Operations and Tasks" in Chapter 6).

UNSPEC (v) The letter v represents a scalar variable of arithmetic, string, or pointer type. The expression on the right is evaluated and converted to a bit string (whose length is an implementation defined function of the characteristics of v), and 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 just large enough to hold the bit string.

ALPHABETIC LIST OF STATEMENTS

The ALLOCATE Statement

Function:

The ALLOCATE statement causes storage to be allocated for specified controlled data.

General format:

Option 1:

```
ALLOCATE [level] identifier
   [dimension] [attribute]...
   [,[level] identifier  [dimension]
   [attribute]...]...;
```

Option 2:

```
ALLOCATE based-variable-identifier
   SET (pointer-variable)
   [IN (area-variable)]
   [, based-variable-identifier
   SET (pointer-variable)
   [IN (area-variable)]]...;
```

Syntax rules:

1. Based variables and nonbased controlled variables may both be specified as identifiers 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 a 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.

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.

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 may appear preceding or following the IN clause.

General Rules:

Rules 1 through 6 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 for arrays and lengths of strings are fixed at the execution of an ALLOCATE statement.

   a. If a bound or length is explicitly specified in an ALLOCATE statement, that bound or length overrides any bound or length given in the DECLARE statement.

   b. If a bound or length is specified by an asterisk in an ALLOCATE statement, that bound or length is taken from the most recent generation of data for the identifier in a previous allocation. In case no such generation exists, the bound or length is undefined.

   c. If a bound or length is not specified in an ALLOCATE statement, it must be specified in the DECLARE statement. The scope of this declaration must include the ALLOCATE statement. The expression from the DECLARE statement is evaluated at the point of allocation.

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. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, only 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. To determine whether or not storage has been allocated for an identifier the built-in function ALLOCATION may be used.

5. A parameter that is declared CONTROLLED may be specified in an ALLOCATE statement if the associated argument is given the CONTROLLED attribute and no dummy is created. (See "Relationship of Arguments and Parameters," in Chapter 10).

6. The evaluations implied by the ALLOCATE statement are subject to the same rules as the evaluations involved in prologue activity (see "Prologues," in Chapter 10).

Rules 7 through 15 apply only to Option 2:

7. When Option 2 is used, storage is not "pushed down" or stacked. In this case, reference may be made to any generation of a based variable through a pointer variable.

8. A SET clause must appear with the based variable in the ALLOCATE statement. This clause indicates the pointer variable that is to receive the pointer value identifying the generation for which storage is to be allocated. The SET clause need not name the pointer variable which was declared with the based variable.

9. If the IN clause appears in the ALLOCATE statement, storage will be allocated in the area corresponding to the

specified area variable for the generation of the based variable. If sufficient storage does not exist within this area, the AREA condition will be raised.

10. If the IN clause is omitted, space will be allocated in systems storage for the generation of the based variable.

11. The amount of storage allocated for a based variable depends on its attributes, and on its dimensions and length specifications if these are applicable at the time of allocation. These attributes are determined from the declaration of the based variable, and additional attributes may not be specified in the ALLOCATE statement. If the allocated variable is a structure whose elements are dimensioned variables or variable length strings, and the dimensions or lengths are themselves defined as elements in the structure, then the dimensions or lengths are taken from that previous generation of the structure defined by the pointer variable named in the DECLARE statement for that structure. In subsequent references to such allocated variables, calculation of dimensions or string lengths will be made by use of the generation identified by the declared pointer. Note, however, that the asterisk notation for bounds and length is not permitted for based variables (see "The CONTROLLED Attribute" in Chapter 4).

12. If the area variable is an array, the subscripts must be specified with the area variable.

13. A based variable transferred as an argument to a procedure may not appear in an ALLOCATE statement in the called procedure.

14. The pointer value defined at the first allocation into an area variable is not necessarily equivalent to a pointer value defined by the ADDR (area-variable) function.

15. If the INITIAL attribute is specified in the declaration of the based variable, the initialization occurs after the allocation of the variable and after the pointer variable has been assigned a value.

Examples:

1. The following examples illustrate the use of the ALLOCATE statement for a nonbased identifier:

```
DECLARE A(N1,N2) CONTROLLED;

N1, N2 = 10;
ALLOCATE A;          The bounds are 10 and
                       10
ALLOCATE A           The bounds are K1 and
    (K1,K2);           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           The bounds are J1 and
    (J1, J2);          J2.
```

2. The following example illustrates the use of the ALLOCATE statement when the DECLARE statement contains asterisks for the length of a nonbased bit string B:

```
DECLARE B BIT (*)  VARYING CONTROLLED;

ALLOCATE B           Illegal; violates rule
    BIT (*);           2b.
ALLOCATE B;          Illegal; violates rule
                       2b.
ALLOCATE B           The maximum length is
    BIT (N);           N.
ALLOCATE B CHAR-     Illegal; violates syn-
    ACTER (4);         tax rule 5.
ALLOCATE B           The maximum length is
    BIT (8);           8.
```

3. The following example illustrates the use of the built-in function ALLOCATION and of the INITIAL attribute for a nonbased identifier 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 CONTROLLED (P),
    RATES (I) CONTROLLED (Q),
    1 GROUP CONTROLLED (R),
      2 PTS (J) POINTER,
      2 VALUES (J) FIXED,
    TABLE AREA STATIC EXTERNAL,
        S POINTER;

ALLOCATE VALUE SET (P);
   Allocates space in systems storage
   for a generation of the based vari-
```

able 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 dimensions of each of the components PTS and VALUE are determined by the value of J.

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 expressions and to assign values to scalars, arrays, and structures.

General format:

Option 1. (Scalar Assignment)

```
{scalar-variable}  [,scalar-variable  ]...=scalar-expression;
{pseudo-variable}  [,pseudo-variable  ]
```

Option 2. (Array Assignment)

```
{array       } [,array       ] ...
{pseudo-array} [,pseudo-array ]
    ={array-expression [,BY NAME];}
     {scalar-expression;          }
```

Option 3. (Structure Assignment)

```
{structure       }[,structure       ]...
{pseudo-structure}[,pseudo-structure]
    =structure-expression [,BY NAME];
```

Option 4. (Statement Label Assignment)

```
scalar-label-variable
   [,scalar-label-variable]...=
        {label-constant;       }
        {scalar-label-variable;}
```

```
array-label-variable [,array-label-
   variable]...=
        {label-constant;       }
        {scalar-label-variable;}
        {array-label-variable; }
```

Option 5. (Pointer Assignment)

```
pointer-variable
     [,pointer-variable]...=
          pointer-expression;
```

```
array-pointer-variable
   [,array-pointer-variable]...=
            {pointer-expression     }
            {array-pointer-variable};
```

Syntax rules:

1. In Option 1, each variable on the left of the equal sign may be of arithmetic, bit, or character data type.

2. In Option 2, each array referred to on the left of the equal sign may be an array variable name or a pseudo-array. If the BY NAME option is present, those arrays must be arrays of structures. A pseudo-array is a pseudo-variable whose arguments are array variable names. (In the case of the pseudo-variable SUBSTR (s,i,k), this requirement applies only to the argument s; see " Pseudo-Variables.")

   All of the arrays on the left and the arrays in the array expression must have the same number of dimensions and identical dimension bounds.

   If a scalar expression appears to the right of the equal sign, the value of this expression is assigned to every element of the array on the left.

   If the expression to the right of the equal sign contains structure operands, all arrays in the statement must be arrays of structures. If the BY NAME option is not used, the structuring of the structure operands must be equivalent to the structuring of the structures in the arrays of structures.

3. In Option 3, in the absence of the BY NAME option, the structure indicated on the left must have structuring identical to the structures indicated in the structure expression. Actual level numbers of the structures involved need not be the same; only the structuring described need be the same.

General rules:

1. The assignment statement is evaluated as follows:

   a. In Options 1, 4, and 5, if any expressions appear on the left of the equal sign, either in subscripts or in pseudo-variables, these expressions are evaluated exactly once from left to right. The expression on the right of the equal sign is evaluated. The value of the expression on the right of the equal sign is assigned to the variables on the left of the equal sign, from left to right.

   b. In Options 2 and 3, the assignment statement is treated as if it were a sequence of scalar assignment statements applied on an element-by-element basis. See Rules 3 and 4 below for a discussion of the evaluation of a structure or array assignment BY NAME.

   c. The definition of the order of assignment for a statement of the form

      L1: A,B=expression;

      (where A and B are arrays of dimensionality n) is as follows:

```
L1:  DO I1 = LBOUND (A,1) TO HBOUND (A,1);
     DO I2 = LBOUND (A,2) TO HBOUND (A,2);
        .
        .
        .
     DO In = LBOUND (A,n) TO HBOUND (A,n);
     A(I1, I2,...,In), B(I1,I2,...,In) =
        expression;
```

      Subscripts (I1,..., In) are inserted for the appropriate arrays on the righthand side, thus yielding a sequence of scalar assignments.

      The result of the evaluation for a later position in an array or structure may be affected by the evaluation and assignment to an earlier position (see Example 1, below).

   d. When necessary, the expression value, or values, is converted to the characteristics of the variable on the left according to the rules in "Expressions," in Chapter 3, except when conversion of arithmetic base is involved (this is converted directly to the precision of the variable to the left of the equal sign).

   e. Structure assignment, in the absence of the BY NAME option, is accomplished through the following process:

      Consider that each structure identifier designates a structure having n elements at the next level. The structure assignment statement is transformed into n statements, S_1, S_2, ..., S_n, with each statement S_i involving the ith element of each structure (see example 4 below).

2. When a variable on the left is a bit or character string or the UNSPEC pseudo-variable, the expression is evaluated as above, and the assignment is performed from left to right, starting with the leftmost position.

   a. If the string has a fixed length and the value of the expression is longer than the string, the value is truncated at the right.

   b. If the string has a fixed length and the value of the expression is shorter than the string, the value is extended on the right with zeros for bit strings or with blanks for character strings.

   c. If the string has a varying length and the value of the expression is longer than the maximum length of the string, the value is truncated; the assigned string is of the maximum length.

   d. If the string has a varying length and the value of the expression is shorter than the maximum length of the string, the value is assigned; the new length of the string is the length of the value.

   e. If the variable on the left is the pseudo-variable SUBSTR with an argument that is a varying-length string, the assignment is performed to this substring in precisely the same way as it would be if the argument were of fixed length, where this fixed length is the length defined by the SUBSTR pseudo-variable.

3. If the BY NAME option is used for arrays of structures in Option 2, the assignment statement is treated as a sequence of BY NAME structure assignments applied on an element-by-element basis.

4. If the BY NAME option is used in Option 3, the assignment statement is evaluated as follows:

   a. Every element at the next level of each structure is extracted.

   b. A subset of these elements is selected. This subset consists of those elements common to all of the structures.

   c. A corresponding assignment statement is constructed for each of the subset elements. The order of the constructed statements corresponds to the order in which the elements appear in the leftmost structure. The rules by which such statements are constructed are detailed in paragraphs d, e, and f below.

   d. If all of the elements corresponding to a subset element are structures or arrays of structures, an assignment statement is constructed and the BY NAME option is appended to it. (Further statements are generated from this constructed statement in accordance with the rules given in paragraphs 4a through 4f.)

   e. If none of the elements corresponding to a subset element is a structure or an array of structures, an assignment statement is constructed but the BY NAME option is not appended to it. No further statements would be generated from this constructed statement.

   f. If the rules in paragraphs d and e above do not pertain, no statement is constructed.

Example:

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
```

Note that the structures contain array names.

According to the rule stated in paragraph 4a, the elements extracted are as follows:

```
ONE.PART1        TWO.PART1
ONE.PART2        TWO.PART2
ONE.PART3

          THREE.PART1
          THREE.PART2
```

As indicated by the rule given in paragraph 4b, a subset of those elements common to all of the structures is then selected. This subset is

```
PART1
PART2
```

If the following statement were being evaluated,

```
ONE = TWO-2*THREE, BY NAME;
```

then the following statements would be constructed (see 4c and 4d):

```
ONE.PART1 = TWO.PART1-2*
            THREE.PART1, BY NAME;

ONE.PART2 = TWO.PART2-2*
            THREE.PART2, BY NAME;
```

Further statements are generated in accordance with the rules in paragraphs 4a through 4f until the lowest level is reached.

Note: In BY NAME structure assignment, it is unnecessary for the structuring of all participating structures to be identical. Names of variables that are defined on structures appearing in BY NAME assignment take no part in name matching (see "The DEFINED Attribute").

5. In Option 4, the value of the label constant or the label variable is qualified by an identification of the current invocation of the block containing the label and by the current task.

   This qualification information is used when a GO TO statement specifies

the label variable to make the identified invocation current and to check that control does not cross task boundaries.

6. Pointer variables may be components of structures or arrays of structures, in which case they are assigned values by a statement as specified in Options 2 and 3. However, no conversions are performed, and the value assigned to a pointer structure component must be a pointer variable. If the pointer variables are array pointer variables, the rules for array assignment given in Rule 1 apply. In any event, the pointer expression is limited to a scalar pointer variable or a function reference that returns a scalar pointer value.

Examples:

1. 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.

2. 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'.
```

3. The following examples illustrate scalar assignment (Option 1):

a. A,B,C = A+SIN(B) + C**2; provided X has the characteristics of the expression, this is the same as

```
X = A+SIN(B) + C**2;
        A = X;
        B = X;
        C = X;
```

b. COMPLEX (U1, V1) = COMPLEX (U, V) + REAL (Q);

This is the same as

```
C=COMPLEX(U,V)+REAL(Q);
U1=REAL(C);
V1=IMAG(C);
```

4. The following examples illustrate structure assignment (Option 3):

a.
```
DECLARE 1X, 2Y, 2Z, 2R, 3S, 3P,
1A, 2B, 2C, 2D, 3E, 3Q;
   X = X*A;
```

The second statement is equivalent to the following statements:

```
Y = Y*B;
Z = Z*C;
S = S*E;
P = P*Q;
```

b.
```
DECLARE 1A, 2B, 2C, 3D, 3E;
   A = A+B;
```

The second statement expands into the following:

```
   B = B+B;
   C = C+B;
```

The last statement expands into

```
   D = D+B;
   E = E+B;
```

5. The following example illustrates statement label assignment (Option 4);

```
DECLARE P LABEL;
     P = A;
     GO TO P;
     .
     .
     .
     A: X = Y**2;
```

This set of statements causes control to transfer to A when the GO TO P statement is executed.

6. The example below illustrates assignment to an array of structures (Options 2 and 3).

   In the following statement, A is an array of structures, and R is a structure:

```
DECLARE 1A(2,2), 2B, 2C, 2D, 3E, 3F,
        1R, 3S, 3T, 3U, 5V, 5W;
```

   The following is an array assignment statement:

```
A=R;
```

   The above assignment statement is equivalent to the following four structure assignment statements:

```
A(1,1)=R;
A(1,2)=R;
A(2,1)=R;
A(2,2)=R;
```

   The four statements above are, in turn, equal to the following:

```
A(1,1) .    B,   A(1,2).B,   A(2,1).B,
    A(2,2).  B=S;

A(1,1).C, A(1,2).C, A(2,1).C,  A(2,2).
    C = T;

A(1,1).E,  A(1,2).E, A(2,1).E, A(2,2).
    E = V;

A(1,1).F, A(1,2).F, A(2,1).F, A(2,2).F
    = W;
```

   (If R is ABNORMAL, 16 statements are actually generated.)

7. The following example illustrates conversion of data defined by a picture description assigned to floating-point data, and vice versa:

```
DECLARE A FLOAT, B PICTURE '999V99';

A=B; (B is converted from fixed-point
          to floating-point.)

B  = A; (A is converted from floating-
          point to fixed-point.)
```

8. The following example illustrates pointer assignments (Option 5):

```
DECLARE  (P, Q(5), R, T(5) ) POINTER,
          VALUE FIXED STATIC,
          POINT ENTRY (FIXED) RETURNS
          (POINTER);
P=R;
R=ADDR (VALUE);
Q(3)=NULL;
T=Q;
Q=ADDR (R);
T(1)=POINT (VALUE);
```

The BEGIN Statement

Function:

The BEGIN statement is the heading statement of a begin block.

General format:

```
BEGIN;
```

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): 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 (expression)];
```

Syntax rules:

1. The entry name represents the entry point of the procedure invoked.

2. Each argument may be any of the following: any type of expression, a statement label constant, a statement label variable, a statement label array, a label parameter, an entry name, an entry parameter, a file name, a file parameter, a task name, a task parameter, an event name, an event parameter, an area name, an area parameter, a pointer name, a pointer expression, or a pointer parameter. Note that a pointer expression must be either a pointer variable or a pointer function reference.

3. The TASK, EVENT, and PRIORITY options can appear in any order. They are separated from each other by blanks, and they are separated from the initial part of the CALL statement by a blank.

4. The scalar event and task names may be subscripted references to event or task arrays.

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. Note that if either the EVENT option or the PRIORITY option, or both, are used without the TASK option, the created task will have no name (see "Asynchronous Operations and Tasks" in Chapter 6).

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. The value of the completion status for the event name (i.e., the value of EVENT (event name)) is set to '0'B on execution of the CALL statement and to '1'B on completion of the created task. (see "Event Data" in Chapter 2 and "The WAIT Statement" in this chapter.)

4. If the PRIORITY option is used, the expression in the above form is evaluated when the CALL statement is executed. The result of this evaluation is converted to FIXED (m,o) where m is implementation-defined. The priority of the named task is then made m relative to the task in which the CALL is executed. If the PRIORITY option is not specified, a priority must have been assigned at some earlier point through the PRIORITY pseudo-variable.

5. See "Relationship of Arguments and Parameters" for a detailed description of the interaction of CALL arguments and invoked entry parameters.

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. It also dissociates from the specified file, all of the attributes declared for it in the opening of that file (thus, if so desired, the file name may be respecified with new attributes in a subsequent OPEN statement). However, all declared attributes for that file (i.e., all attributes explicitly given in a DECLARE statement) remain in effect.

General format:

```
CLOSE options-group [,options-group]...;
```

Following is the format of "options group":

FILE(filename) [IDENT(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. A CLOSE statement unlocks all records previously locked in the task in which the CLOSE appears.

8. 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 string. 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 to the character string variable.

Output files: The argument is an expression; this is evaluated and converted to a character string, which 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 string. 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, for a discussion of the DECLARE statement.

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 rule:

Execution of the DELAY statement causes the scalar expression to be evaluated and converted to an integer n and execution 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);

Execution of the controlling task is suspended for ten milliseconds.

The DELETE Statement

Function:

The DELETE statement deletes a record from a DIRECT UPDATE file.

General format:

DELETE option-list ;

Following is the format of "option list":

```
FILE(filename) KEY(expression)
   [EVENT(event-variable)]
```

General rules:

1. The options may appear in any order.

2. The FILE(filename) option specifies the UPDATE file; it must occur once.

3. The KEY(expression) option specifies the key that identifies the record to be deleted. This option must occur once.

4. If the EVENT(event variable) option is given, the execution will not wait for the deletion to be completed before continuing with subsequent statements. The event variable will be given the value '0'B until the deletion is complete, when it will be given the value '1'B.

5. The DELETE statement unlocks a record only if that record had been locked in the same task in which the DELETE appears.

6. 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 (character-variable)
[EVENT (event-variable)];
```

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 program 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 event variable will be given the value '0'B until the reply is received, when it will be given the value '1'B.

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 (see "Groups") and may specify iterative execution of the statements within the group.

```
Option 1

                                DO;

Option 2.

              DO WHILE (scalar-expression);

Option 3.

              DO { pseudo-variable } = specification [,specification]...;
                 { variable      }

              A specification has the following format:

                 [ TO expression2     [BY expression3] ]
expression 1     [                                    ] [WHILE (expression 4)]
                 [ BY expression3     [TO expression2] ]
```

Figure 4. General Format for the DO Statement

Syntax rules:

1. The "variable" in Option 3 is a subscripted or unsubscripted scalar variable. Label variables, string variables, and complex variables are allowed, provided the expansions given below result in valid PL/I programs.

2. Each "expression" in the specification list is a scalar expression.

3. If BY expression3 is omitted from the specification, expression3 is assumed to be one (1).

4. If TO expression2 is omitted from the specification, the iteration is performed indefinitely until terminated by the WHILE clause or by some other statement within the scope of the DO.

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.

6. If the variable in Option 3 is a label variable, each specification must take the form:

```
{ label-variable }
{                } [WHILE (expression4)]
{ label-constant }
```

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 an iteration 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 iteration defined by the following:

```
LABEL: DO variable = expression1
         TO expression2 BY expression3
         WHILE (expression4);
       statement 1
       .
       .
       .
       statement n
LABEL1:END;
NEXT:  statement
```

The above is exactly equivalent to the following expansion:

```
LABEL:t1=sexp1; t2=sexp2;...; tm=sexpm;
      e1=expression1; e2=expression2;
      e3=expression3;
      v=e1-e3;
LABEL1: v=v+e3; IF (e3>=0)&(v>e2)|(e3<0)&
        (v<e2)
      THEN GO TO NEXT;
IF (expression 4) THEN; ELSE GO TO NEXT;
      statement 1
      .
      .
      .
      statement n
      GO TO LABEL1;
NEXT: statement
```

In this expansion sexp1,...,sexpm are the expressions which appear in subscripts of the control variable or pseudo-variable, followed by the second and third argument positions if the SUBSTR pseudo-variable is being used. The letter v denotes the control variable with all sexpi replaced by ti. In the simplest cases, m=o and the first statement is e1=expression1. The variables t1,...,tm, are BINARY FIXED integer variables of default precision, inserted by the compiler. The variables e1, e2, and e3 have the characteristics of the corresponding expressions.

a. If more than one specification is given, the statement labeled NEXT refers to the initialization for the next specification; for example:

```
NEXT:  e5 = expression 5;
```

Note: Each specification applies to the statements in the DO group. The ti variables are computed only once per DO group.

b. If the WHILE clause is omitted, the IF statement involving expression4 and the ELSE GO TO NEXT statement are deleted.

c. If the TO clause is omitted, the IF statement and the assignment statement involving e2 are omitted.

d. If both the TO clause and the BY clause are omitted, all statements involving e2 and e3 are omitted as well as the statement "GO TO LABEL1;".

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 uninterrupted. 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 transfer into a DO group from outside the DO group only if the DO group is delimited by the DO statement in Option 1; that is, iteration is not specified.

7. The effect of allocating or freeing the control variable within the DO loop 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 groups.

General format:

```
END [label];
```

General rules:

1. If a label follows END, the END statement terminates that group or block having that label.

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. An END statement may be used to terminate more than one group or block (see "Use of the END Statement," in Chapter 1).

4. If control reaches an END statement, terminating a procedure, it is treated as a RETURN statement.

5. If control reaches an END statement which terminates a BEGIN block that is an on-unit, control is returned to the point following the interrupt location.

6. 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-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 (see "Relationship of Arguments and Parameters").

2. The data attributes permitted with a PROCEDURE statement are the arithmetic, string, and pointer attributes. The data attributes specify the characteristics of the value returned by the procedure when invoked as a function at this entry point. (This rule applies to each entry name by which the entry point may be invoked.) The value specified in the RETURN statement of the invoked entry is converted, if necessary, to have the specified data attribute.

   If data attributes are not completely specified at the entry point, default attributes are applied, as determined by the name of the entry point.

   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 and data attributes.

Consider the statement:

```
A:I: ENTRY;
```

This statement is equivalent to:

```
A:  ENTRY;
I:  ENTRY;
```

The ENTRY statement must be internal to the procedure block for which it defines a secondary entry point. The ENTRY statement may not be internal to any block contained in this procedure; nor may it be within a DO group that specifies iteration.

Example:

```
NAME:     PROCEDURE(N) CHARACTER(15);
          DECLARE TABLE(100) CHARACTER(15)
          EXTERNAL;

INITIAL:  ENTRY(N) CHARACTER(1);
          RETURN (TABLE(N));
          END;
```

The EXIT Statement

Function:

The EXIT statement causes immediate termination of the task that contains the statement and all tasks attached by this task (see "Asynchronous Operations and Tasks," in Chapter 6). If the EXIT statement is executed in a major task, it is equivalent to a STOP statement (see this chapter).

General format:

```
EXIT;
```

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.

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 nonbased controlled variables to be freed. For nonbased variables, the next most recent allocation is made available, and subsequent references to the identifier refer to that allocation.

General formats:

Option 1

```
FREE identifier [,identifier] ...;
```

Option 2

```
FREE [pointer-variable->]
    based-variable-identifier
    [,[pointer-variable->]
    based-variable-identifier]...;
```

Syntax rule:

Each identifier is a scalar, array, or major structure name of the controlled (based or nonbased) storage class.

General rules:

1. The freeing of nonbased and based controlled variables may be specified in the same FREE statement.

2. Controlled storage allocated in a task cannot be freed by a descendant task.

3. If a specified nonbased identifier has no allocated storage at the time the FREE statement is executed, it is an error.

Rules 4 through 6 apply only to Option 2.

4. If the based variable is not qualified by pointer qualification, the pointer declared with the based variable will be used to identify the generation of data occupying the portion of storage to be freed (see Chapter 2 for a discussion of pointer qualification).

5. The amount of storage freed depends upon the attributes of the based variable, including bounds and/or lengths at the time the storage is freed, if applicable. The user is responsible for determining that this amount coincides with the amount allocated. If the variable had not been allocated, the results are unpredictable.

6. Based variables appearing in ALLOCATE statements with the IN clause may not appear in FREE statements. In other words, allocations within storage areas defined by area variables may not be freed.

Examples:

1. FREE X,Y,Z;

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;
                .
                .
                .
X=SIN(C**2 + X/Y);
```

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) | STRING(character-
  string-name) ]
data-specification [COPY]
```

General rules:

1. If neither the FILE(filename) option nor the STRING(character-string-name) option appears, standard system input file SYSIN is assumed.
2. One data specification must appear.
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 character string name refers to the character string which is to provide the input data. 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 specified by the data specification, the ERROR condition is raised.
6. The data specification is as described in Chapter 7.
7. If the FILE(filename) option refers to an unopened file, the file is opened automatically; the effect is as if the GET statement were preceded by an OPEN statement referring to the file.
8. The COPY option, which may only be used with the FILE(filename) option, specifies that the source data, as read, is to be written, without alteration, on the standard system print file SYSPRINT.

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.

The GO TO Statement

Function:

The GO TO statement causes control to be transferred to the specified statement.

General format:

```
{GO TO}{label-constant;          }
{GOTO }{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 may not pass control to an inactive block (see "Activation and Termination of Blocks," in Chapter 6, for a discussion of active and inactive blocks).

   A GO TO statement may not transfer control from outside a DO group to a statement inside the DO group if the DO group specifies iteration unless the GO TO terminates a procedure invoked 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. Conditions are reinstated, and automatic variables are freed in the same way as if the blocks 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 may not terminate any procedure invoked during a prologue (see "Prologues" in Chapter 10), or an ALLOCATE statement.

5. A GO TO statement may not be used to transfer control from a task to its attaching task or to any of its descendant tasks.

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 causes program flow to depend on the value of an expression.

General format:

```
IF scalar-expression THEN unit-1 [ELSE
   unit-2]
```

Syntax rules:

1. Each "unit" is either a group or a begin block, either of which would be terminated by a semicolon.

2. The IF statement is not itself terminated by a semicolon.

3. Each unit may be labeled.

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, the unit-1 is executed, and control passes to the statement following the IF statement. If all bits have the value 0, the 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 (see "Sequence 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. Since

each ELSE clause is always associated with the innermost preceding IF, an ELSE with a null statement may be required to specify a desired sequence of control.

Examples:

1. IF QUEUE = EMPTY THEN CALL COMPILE;
   ELSE GO TO MULTIPROCESS;

2. A: IF X > Y THEN
      IF Z = W THEN
      IF W < P THEN Y = 1;
      ELSE P = Q;
      ELSE;
   ELSE X = 4;
   J: Z = 5;

The LOCATE Statement

Function:

The LOCATE Statement, which applies to BUFFERED OUTPUT files, allows a record to be created in buffer storage and subsequently written (see "The Buffering Attributes", Chapter 4).

General format:

```
LOCATE variable option-list ;
```

Following is the format of "option list":

```
FILE(filename) SET(pointer-variable)
   [KEYFROM(expression)]
```

General rules:

1. The options in the option list may appear in any order.

2. The "variable" must be an unsubscripted level 1 based variable and it cannot contain VARYING length strings.

3. The FILE(filename) option specifies the file involved. This option must appear.

4. The SET(pointer-variable) option specifies a subscripted or unsubscripted POINTER variable which is to be set to identify the variable in the buffer. This option must appear.

5. If the KEYFROM(expression) option appears, the value of the expression is converted to a character string and included as the key of the record to be subsequently written.

6. The based variable is allocated in a buffer, and the POINTER variable in the SET option is set to identify it. The record identified is written into the output file immediately before the next WRITE, LOCATE, or CLOSE operation on the file, at which time the record is freed.

7. If the FILE(filename) option refers to an unopened file, the file is opened automatically; the effect is as if the LOCATE statement were preceded by an OPEN statement referring to the file.

Example:

```
LOCATE ALPHA SET (REC_POINT) FILE
   (BETA);
```

The based variable ALPHA is allocated in a buffer and the pointer variable 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.

The Null Statement

Function:

The null statement causes no action and does not modify sequential operation.

General format:

```
[label:]...;
```

Example:

```
      .
      .
      .
ON OVERFLOW;
      .
      .
      .
```

The on-unit (see "The ON Statement") 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, FORMAT, PROCEDURE, or DECLARE) or an unlabeled begin block. 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 appear within the begin block.

General rules:

1. The standard action to be taken for all ON-conditions is established 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.

2. 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.

3. If SNAP is specified, then when the given condition occurs, a calling trace is listed.

4. Control can reach an on-unit only when an interrupt occurs for the condition associated with this on-unit in an ON statement.

5. If an action specification is established by an ON statement in a given block, it remains in effect throughout this block and throughout all dynamic descendants of this block (see "Activation and Termination of Blocks," in Chapter 6, for a discussion of blocks and generations of blocks).

   If an action is specified more than once in a given block, the effect of the old (or prior) ON statement is either temporarily suspended or completely nullified by the new (or later) ON statement, as follows:

   a. If the new (or later) ON statement is in a block dynamically descended from the block containing the old (or prior) ON statement, the effect of the old ON statement is temporarily suspended or stacked. The effect of the old ON statement is restored by execution of a REVERT statement or upon termination of the block containing the new ON statement.

   b. If the new (or later) ON statement and the old (or prior) ON statement are internal to the same invocation of the same block, the effect of the old ON statement is completely nullified.

6. If an action is specified by an ON statement in a particular task, the effect of this ON statement is inherited by each attached task and by each task attached by the attached task, etc. (see "Asynchronous Operations and Tasks," in Chapter 6 , for a discussion of attached and attaching tasks).

7. A condition raised during execution results in an interrupt if and only if the condition is enabled at the point where it is raised.

   a. The conditions OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, the input/output conditions, and the conditions CONDITION, FINISH, and ERROR are enabled by default.

   b. The conditions SIZE, SUBSCRIPTRANGE, and CHECK are disabled by default.

   c. The enabling status of OVERFLOW, FIXEDOVERFLOW, UNDERFLOW, ZERODIVIDE, CONVERSION, SIZE, SUBSCRIPTRANGE, and CHECK are controlled by the condition prefix (see "Prefixes" in Chapter 1).

8. A single statement on-unit may not refer to a remote format specification through edit-directed transmission.

9. The identifier list of a CHECK condition, the filename of an input/output

condition, and the on-unit of an ON statement belong to the scope of the procedure or begin block to which the ON statement is internal.

The action specification established by executing an ON statement in a given block remains in effect throughout this block and throughout all dynamic descendants of this block, unless overridden by the execution of another ON statement. Names in an on-unit do not belong to the scope of the dynamic environment at the point of execution of the on-unit, but rather to the environment of the ON statement.

Examples:

1.
```
IOPR:  PROCEDURE;
        .
        .
        .
  R1:   GET FILE (FILEX) EDIT(A,B)
          (2F(7,3));
        ON CONVERSION
          CONVQ = 9999;
        .
        .
        .
  R2:   GET FILE (FILEX) EDIT (X)
          (A(6));
        END IOPR;
```

Assume that program execution begins with procedure IOPR.

If an illegal character is read from FILEX during the execution of statement R1, the standard system action occurs.

The ON statement specifies that the execution of the statement CONVQ = 9999 is to occur in the event that a conversion error causes an interrupt subsequent to execution of the ON statement. Thus, if a conversion error occurs during the transmission of X in statement R2, the normal sequence of control is interrupted, and the statement CONVQ = 9999 is executed.

2.
```
ZCHK:  PROCEDURE;
        .
        .
        .
  S1:   ON OVERFLOW OVSWCH = 1;
        .
        .
        .
        CALL Q;
        .
        .
        .
   Q:   PROCEDURE;
        .
        .
        .
  S2:   ON OVERFLOW OVSWCH = 2;
        .
        .
        .
  S3:   ON OVERFLOW SYSTEM;
        .
        .
        .
        END Q;
      END ZCHK;
```

Assume that program execution begins with procedure ZCHK.

If an overflow occurs prior to execution of the S1 statement, an interrupt with standard system action occurs. If an overflow occurs subsequent to execution of the S1 statement, an interrupt occurs, and the statement OVSWCH = 1 is executed.

When procedure Q is invoked, the S1 statement remains in effect until the S2 statement is executed. At this point, the effect of the S1 is temporarily suspended, and the S2 goes into effect.

If an overflow occurs between S2 and S3, an interrupt occurs, and the statement OVSWCH = 2 is executed.

When S3 is executed, it completely replaces S2 (S1 is still stacked). If an overflow occurs after S3 is executed and before the end of procedure Q, it causes the standard system action to take place.

After control is returned from Q to ZCHK, S3 is completely replaced by S1, whose effect is restored. Any overflows occurring from this point to the end of procedure ZCHK cause the statement OVSWCH = 1 in S1 to be executed.

3.
```
SBCHK:  PROCEDURE;
        DECLARE A(9);
B1:  . . .A(I)...;
        ON SUBSCRIPTRANGE  BEGIN;
                           IF I>9 THEN
                            GO TO BIGER;
                           ELSE GO TO
                            LITLER;
                           .
                           .
                           .
                     BIGER:  ...;
                           .
                           .
                           .
                     LITLER: ...;
                           END;
(SUBSCRIPTRANGE): B2:...A(I)...;
        B3:...;
          END SBCHK;
```

Assume that procedure SBCHK is the only procedure in the program.

At the beginning of execution, any occurrence of the condition SUBSCRIPTRANGE will not give an interrupt; it is not enabled, since the condition name does not appear in a prefix in the PROCEDURE statement. If in statement B1, the value of I is greater than 9 or less than 1, no interrupt action is taken.

When the ON statement for the condition SUBSCRIPTRANGE is executed, any interrupt that results from a subsequent occurrence of the SUBSCRIPTRANGE condition will result in the action specified by the begin block in the ON statement.

The prefix for statement B2 specifies that the condition SUBSCRIPTRANGE is enabled and should cause an interrupt if it occurs during the execution of statement B2. In this case, the begin block in the ON statement is executed.

In the execution of B3 and subsequent statements, the occurrence of a subscript that is not within the specified range does not cause an interrupt action to occur.

For further examples, see "Interrupt Operations" in Chapter 6.

The OPEN Statement

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(argument)]
[TITLE(expression)]
[INPUT | OUTPUT | UPDATE][STREAM |
RECORD]    [DIRECT | SEQUENTIAL]
[BUFFERED | UNBUFFERED]    [EXCLUSIVE]
[KEYED(decimal-integer-constant)]
[BACKWARDS] [PRINT]
[LINESIZE (expression)]
[PAGESIZE (expression)]
```

General rules:

1. The INPUT, OUTPUT, UPDATE, STREAM, RECORD, DIRECT, SEQUENTIAL, BUFFERED, UNBUFFERED, EXCLUSIVE, KEYED, BACKWARDS, and PRINT options specify attributes which augment the attributes specified in the file declaration; for rules governing which of these attributes can be applied together, see "File Description Attributes," in Chapter 4, and "File Opening and File Attributes," in Chapter 7.

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. Opening an already open file does not affect the file if the second opening takes place in the same task or in an attached task. Expressions in the options groups are evaluated but not used.

5. The "argument" in the IDENT option is used as follows:

   Input files: The argument must be a character-string variable and 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.

   Output files: The argument is an expression; this is evaluated and converted to a character string which is

placed with the data set as a header label.

Update files: The argument must be a character-string variable and 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 identifies the data set to be associated 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.

7. The LINESIZE option can be specified only for a STREAM PRINT file. The expression is evaluated, converted to an integer, and used as the length of a line during subsequent output to the file. New lines may be started by use of the printing and control format items or PUT statement options, in which case the current line is filled to its full length with blanks. If a line becomes overfilled before action to start a new line is taken, characters spilling over are put onto the next line automatically. Default is implementation-defined.

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 as WORKFILE, whereas ALPHA is associated with a data set named ALPHA.

2. OPEN FILE (MASTER) UPDATE;

   The file MASTER is opened as an UPDATE file. MASTER is taken as the name of the data set.

The PROCEDURE Statement

Function:

The PROCEDURE statement has the following functions:

1. Heads 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 attributes may appear in any order and are separated by blanks.
2. The attributes in the OPTIONS list are separated by commas, where necessary.

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 "Relationship of Arguments and Parameters," in Chapter 10).
2. The OPTIONS attribute specifies a list of options, separated by commas where necessary. The list depends upon implementation. The OPTIONS attribute may be specified only for an external procedure. If specified, it applies to all of the entry points that the procedure might have.
3. The RECURSIVE attribute specifies that this procedure may be invoked recursively. This attribute applies only

to the procedure for which it is declared, and as a result applies to all of the entry points for that procedure.

4. The data attributes permitted with a PROCEDURE statement are the arithmetic, string, 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 appended to the PROCEDURE statement.) The value specified in the RETURN statement of the invoked procedure is converted to the specified data attributes.

   If data attributes are not specified, or if an incomplete set of data attributes is given at the entry point, default attributes are supplied. In the first case, the name of the entry point is used to determine the default base and scale.

   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 specified in the PROCEDURE statement.

For example, the statement:

```
A:I:   PROCEDURE;
```

is equivalent to:

```
A:  PROCEDURE;
I:  ENTRY;
```

Example:

```
B:  PROCEDURE;
          .
          .
          .
    C=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 C in procedure B.

The PUT Statement

Function:

The PUT statement normally causes values of specified variables to be assigned to data fields in a data set. Alternatively, the data may be assigned to a character string variable.

General format:

```
PUT option-list ;
```

Following is the format of "option list":

```
[FILE(filename)          |
  STRING(character-string-name) ]

[data-specification][PAGE]
  [SKIP [(expression)]]
  [LINE  (expression) ]
```

General rules:

1. If neither the FILE (filename) option nor the STRING (character string name) appears, the standard system print 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 "character-string name" refers to the character string variable or pseudo-variable that is to receive the values. Each PUT operation using this option always begins at the beginning of the specified string.

   After appropriate conversion, as for a non-PRINT file, the data specified by the data list is assigned to the string starting at the left-most character. Blanks and delimiters are inserted as usual. 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 three options PAGE, SKIP, and LINE may be given only for PRINT files, and they take effect before transmission of any values defined by the data specification, if given. Of the three, only PAGE and LINE may appear together in a 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 pre-

sent, 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. If w is greater than zero, w-1 blank lines are created, and the wth line, relative to the current line, becomes the new current line. If w is not greater than zero, the effect is that of a carriage return with the same current line; characters previously written may be overprinted. If the expression is not present, SKIP (1) is implied. If less than w lines remain on the current page as determined by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised.

7. The LINE option causes a new current line to be defined for the data set. The expression is converted to an integer w. The LINE option specifies that blank lines are to be inserted so that the next line will be the wth line of the current page. If at least 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, 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 an unopened file, the file is opened automatically for output. The effect is as if the PUT statement were preceded immediately by an OPEN statement referring to the file.

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.

The READ Statement

Function:

The READ statement transfers a record from a RECORD INPUT or RECORD UPDATE file to a variable in internal storage.

General format:

```
READ option-list ;
```

Following is the format of "option list":

```
FILE (filename)
      [INTO (variable)          ]
      [SET(pointer-variable)    ]
      [IGNORE(expression)       ]

       [KEY (expression)]
       [KEYTO
        (character-string-variable)]
       [EVENT (event-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 has not been opened, it is opened automatically. The effect is as if the READ statement were preceded by an OPEN statement referring to the file.

3. The INTO(variable) option specifies an unsubscripted level 1 variable in internal storage into which the record is to be read. The variable cannot contain VARYING character strings.

4. 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.

5. The KEYTO(character-string-variable) option may be given only if the file is SEQUENTIAL and keyed. It specifies that the key of the record is to be copied onto the string variable. KEYTO and KEY may not appear in the same READ statement.

6. 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 given the value '0'B until the execution is complete, when it will be given the value '1'B.

7. Any READ statement referring to an EXCLUSIVE file will cause the record to be locked 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, READ NOLOCK, REWRITE, or DELETE statement that specifies the same key, by a CLOSE statement, or by completion of the 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).

8. The SET option specifies that the record is placed in a buffer, and a pointer variable is assigned its identification such that a based variable may be subsequently referred to via that pointer value. The pointer value is valid until the next READ or until the file is closed.

9. 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).

10. 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 and returns control to the invoking procedure. It may also return a value.

General format:

Option 1.

```
RETURN;
```

Option 2.

```
RETURN (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 may be used to terminate a procedure invoked with the task option. If the task invocation involved an EVENT option (see "The CALL Statement," in this Chapter), then the execution of the RETURN statement will cause the completion status of the associated event name to be set to '1'B.

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.

   If the entry point at which the procedure is invoked specifies data attributes, the value of the expression is converted to the implicit or explicit data attributes specified at the entry point, before it is returned.

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 nullify the effect of the most recent previously executed ON statement for that condition in the containing block and to cause the action specification to be reestablished as it was in the immediate, dynamically encompassing block (see "Activation and Termination of Blocks," in Chapter 6).

General format:

```
REVERT condition;
```

Syntax rule:

The "condition" is any ON-condition (see Appendix 3).

General rules:

The execution of a given REVERT statement, specifying a given condition and internal to a given block, has the effect described above only if an ON statement, specifying the same condition and internal to the same block, was executed after the block was activated. If such an ON statement was executed, and if the execution of no other similar REVERT statement has intervened, then the execution of the given REVERT statement does have the effect described above. Otherwise, the REVERT statement is effectively treated as a null statement. Thus, a repeated REVERT statement results in no operation.

Examples:

```
 A:  PROCEDURE;
     .
     .
     .
ON1: ON ZERODIVIDE GO TO ERRSPEC;
     .
     .
     .
     CALL B;
     .
     .
     .
     B:  PROCEDURE;
         .
         .
         .
     ON2:  ON ZERODIVIDE;
           .
           .
           .
           REVERT ZERODIVIDE;
           .
           .
           .
           END B;
     .
     .
     .
ON3: ON ZERODIVIDE SYSTEM;
     .
     .
     .
     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. If division by zero occurs after execution of the REVERT statement and prior to the execution of statement ON3, an interrupt takes place, and control transfers to the statement GO TO ERRSPEC.

After the execution of statement ON3, division by zero causes standard system action to take place.

The REWRITE Statement

Function:

The REWRITE statement refers to an UPDATE file. The purpose of the statement is to replace an existing record in the data set.

General format:

```
REWRITE option-list ;
```

Following is the format of "option list":

```
FILE(filename) [KEY(expression)]
   [FROM(variable)]
   [EVENT (event-variable)]
```

General rules:

1. The options may appear in any order.
2. The FILE(filename) option specifies the file involved. It must appear. If the file is not open, it is opened automatically.
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 variable cannot contain VARYING character strings.
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 given the value '0'B until the execution is complete, when it will be given the value'1'B.
6. If the record rewritten is one that was locked in the same task, it becomes unlocked.
7. The FROM(variable) option must be specified for a DIRECT UPDATE or SEQUENTIAL UNBUFFERED UPDATE file.
8. If the last record was read by a READ statement with the INTO option, REWRITE without a FROM option has no effect on the record in the data set; but if the last record was read by a READ statement with the SET option, the record will be updated by whatever assignments were made.

Example:

```
REWRITE FILE (ALPHA);
```

The last record read from the data set associated with file ALPHA is rewritten into the data set.

The SIGNAL Statement

Function:

The SIGNAL statement simulates the occurrence of an interrupt (see "Interrupt Operations," in Chapter 6, and "The ON Statement"). It may be used to test the action specification of the current ON statement.

General format:

```
SIGNAL condition;
```

Syntax rule:

The condition may be any one of those described in "ON-Conditions," in Appendix 3.

General rules:

1. When a SIGNAL statement is executed, it is as if the specified condition

had actually occurred. The sequence of control through the program is interrupted, and control is transferred to the current ON-unit for the specified condition. After execution of the ON-unit, control normally returns to the statement immediately following the SIGNAL statement.

2. If an ON statement specifies the CONDITION condition, the condition can cause an interrupt only if a SIGNAL statement, specifying this condition, is given.

3. If the condition specified in the SIGNAL statement is disabled, no interrupt occurs, and the statement is equivalent to a null statement.

4. If the condition has no current ON-unit, then the normal system action for the condition is performed.

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 (see "Asynchronous Operations and Tasks," in Chapter 6).

General format:

```
STOP;
```

The UNLOCK Statement

Function:

The UNLOCK statement makes the specified locked record available for operations on the record.

General format:

```
UNLOCK option-list;
```

Following is the format of "option list":

```
FILE(filename) KEY(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, it is opened automatically. The FILE(filename) option must appear.

3. In the KEY(expression) option, the "expression" is converted to a character string and determines which record is unlocked. This option must appear.

4. A record can be unlocked only by the task which locked it.

The WAIT Statement

Function:

The WAIT statement suspends operations in the task where it appears until certain events have been completed.

General format:

```
WAIT (event-name [,event-name]...)
   [(scalar-expression)];
```

Syntax rule:

The event name is as described in "Event Data" in Chapter 2.

General rules:

1. The execution of this statement causes the task in which it is executed to be suspended until, for some or all of the event names in the list above, the condition

   ```
   EVENT (event-name) = '1'B
   ```

   is satisfied. (See "Asynchronous Operations and Tasks," in Chapter 6, "Event Data" in Chapter 2, "Pseudo-Variables," in this chapter, and the description of the EVENT built-in function in Appendix 1.)

2. If the optional expression does not appear, all the event names in the list must satisfy the above condition before the task issuing the WAIT statement can resume.

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 that must satisfy the above condition before the task issuing the WAIT statement can resume. 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 event names in the list, the value is taken to be n. If the statement refers to an array event name, then each of the array elements contributes to the count.

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 task is suspended until the value of EVENT(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(expression)]
   [EVENT(event-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, it is opened automatically.

3. The FROM(variable) option specifies an unsubscripted level 1 variable which is to be written. It must appear. The variable cannot contain VARYING character strings.

4. The KEYFROM(expression) option is converted to a character string and attached to the record as a 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 given the value '0'B until the execution is complete, when it will be given the value '1'B.

Example:

```
WRITE     FILE(BETA)     FROM(UPDATE)
   KEYFROM(UKEY);
```

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.

CHAPTER 9: COMPILE-TIME FACILITIES

INTRODUCTION

Compile-time is generally defined as that time during which a user's source program is compiled, or translated, into an executable object program. Ordinarily, changes to a source program may not be made at this time.

However, with PL/I, the programmer is allowed to exercise some control over his source program at compile-time. This is made possible through a somewhat different approach to compile-time processing. Compile-time as defined in PL/I is a two-stage process:

1. The Processor Stage -- During this stage, the processor scans the user's input for compile-time statements, i.e., statements that instruct the processor to modify the user's source program. These statements are not considered part of the source program, yet they appear in the input freely intermixed with the statements that constitute the source program. The modified source program (the output from this first stage) then serves as the input to the second stage. Note that if no modification is performed, input to the second stage is exactly the same as the source program that constituted the input to the first stage.

2. The Compilation Stage -- During this stage, the compiler takes the output from the first stage and compiles it into an executable object program.

This chapter is primarily concerned with the first stage; very little, if anything, will be said about the actual compilation of a program. Through the means described in this chapter, the processor can be used to perform many functions; among them are the following:

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.

THE PROCESSOR

PROCESSOR INPUT AND OUTPUT

The processor interprets compile-time statements and acts upon the source program accordingly. Input to the processor consists of a character string, called the source text, which contains compile-time statements and source program text, freely intermixed. Compile-time statements are identified by a leading percent sign (%) and are executed upon being encountered by the processor (with the exception of compile-time procedures, which must be invoked in order to be executed). One or more blanks may separate the percent sign from the statement.

The processor also checks the source program text, but only to insure that there are no unmatched comment or character-string delimiters. A percent symbol appearing within a comment or a character string is considered solely as part of the comment or string, respectively.

Output from the processor 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 processor according to the compile-time statements encountered in the source text.

THE PROCESSOR SCAN

The processor 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 processor places the scanned characters into the program text in the same order and form in which they were encountered. In other words, if there are no compile-time statements, the program text is identical to the source text.

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 processor to continue

the scan from a different point in the source text.

2. The executed compile-time statement may specify to the processor 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 processor 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

When an activated variable or an activated procedure name is encountered in the source text (see "The DECLARE Statement" and "The ACTIVATE and DEACTIVATE Statements" for details about activating), its value becomes a candidate for replacement. This value is then rescanned to determine whether or not it, or any part of it, can be replaced 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.

Example:

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).

Compile-time variables, compile-time procedure references, constants, and operators can be included in the value to be assigned to a compile-time variable; compile-time statements cannot be included in such a value. Note, however, that if the user desires to generate operators such as ¬= and /* into the program text, they must be generated as a complete entity. That is, one cannot, for example, have a /A in the source program and expect a % A = '*' statement to generate the comment delimiter /* in the program text. The reason why this cannot be done is that all replacement values are placed into the program text with one blank appended to each end. Thus, the hypothetical case above would result in /b*b (where each b represents a blank) being generated in the program text.

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 in the section "Compile-Time Statements, Groups, and Procedures."

Briefly, the % prefixed to a statement indicates that the action specified by that statement is to be carried out at the time that it is encountered by the processor. 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 processor. I is then incremented by 1 (% I = I+1;), after which the compile-time IF statement instructs the processor 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 variable 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 STATEMENTS, GROUPS, AND PROCEDURES

Note that wherever keywords are shown below, they may be abbreviated as shown in Appendix 4. Also, a comment appearing within a compile-time statement is never written into the program text.

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 processor that this identifier may cause replacement action in the source program (see "The ACTIVATE and DEACTIVATE Statements" for more details). An identifier may cause such action if and only if it has first appeared in a compile-time DECLARE statement; that is, any use of such an identifier before its appearance in a compile-time DECLARE statement is an error.

General format:

```
% [label:]... DECLARE identifier{CHARACTER|
        FIXED|ENTRY[([CHARACTER|FIXED]
        [,[CHARACTER|FIXED]]...)]
        RETURNS({CHARACTER|FIXED})}
        [,identifier{CHARACTER|FIXED|
        ENTRY[([CHARACTER|FIXED]
        [,[CHARACTER|FIXED]]...)]
        RETURNS({CHARACTER|FIXED})}]...;
```

Syntax rules:

1. Commas must separate declarations of separate identifiers within a single %DECLARE statement.

2. The syntax is the same as the syntax for declaring source program PL/I variables and entry names, but only the CHARACTER (with no length specification), FIXED, ENTRY, and RETURNS attributes are allowed.

3. The attributes may be factored.

General rules:

1. No length may 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 processor. Any reference to a compile-time variable or compile-time procedure name encountered before the variable or procedure name has been declared is in error.

3. The scope of all compile-time variables, compile-time procedure names, and labels of compile-time statements is the entire text scanned by the processor, not including any compile-time procedures that redeclare a compile-time identifier. The scope of a declaration in a compile-time procedure is limited to that procedure.

4. If a compile-time procedure is referred to in the source program, then a compile-time ENTRY declaration must have been given for it. If such an ENTRY does not account for any parameters, it is assumed that the compile-time procedure has none. If, however, parameters are accounted for in the ENTRY declaration, the processor will expect to find a parenthesized list of arguments, separated by commas, in the procedure reference.

   Note that each source program argument to a compile-time procedure is interpreted as a character string, and may not contain commas or right parentheses; the first right parenthesis encountered in such an argument list terminates the list. All left parentheses, except the first, are considered as part of the argument list. For example, the argument list (A(B,C)) would be interpreted as two arguments, namely, A(B and C. (Note also that these interpretations apply only to source program references to compile-time procedures; a compile-time reference to a compile-time procedure is interpreted in the usual fashion.)

   When the source program invokes a compile-time procedure, each argument is scanned for possible replacement values. The actual invocation occurs after all replacement activity, if any, has been performed.

   All arguments in a compile-time procedure reference are converted to the type indicated by the corresponding attribute in the ENTRY declaration for that procedure. If an argument does not have a corresponding attribute in the ENTRY declaration, the argument will not be converted. (For an illustration of a source program reference to a compile-time procedure, see the example at the end of the section "The Compile-Time Procedure.")

5. The value returned to the source program by a compile-time procedure is also scanned for replacement values. The type of the value returned must be the same as the type specified in the RETURNS attribute declaration for that procedure.

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;
```

Syntax rules:

1. The operands of a compile-time expression may be optionally signed decimal integer constants, bit string constants, character-string constants, compile-time variables, compile-time procedure references,or references to the SUBSTR built-in function (with the restriction that the first argument must be a compile-time character-string variable or character-string constant). No other data types or built-in functions are allowed.

2. All PL/I operators, except exponentiation (**), are allowed in compile-time expressions.

General rules:

1. Compile-time expressions are evaluated according to the same rules as source program PL/I expressions, with a single exception: for arithmetic operators, only decimal integer arithmetic of precision (N,0) is performed; that is, each operand of an arithmetic operation is considered to have a precision of (N,0) before the operation is performed, and the result of the operation will also have a precision of (N,0). Thus, the result of the assignment statement below is N + 2 blanks and one zero.

```
% DECLARE A CHARACTER;

% A = 3/5;
```

When required, as in the case above, conversions from fixed point to character string, and vice versa, are carried out according to the rules followed for such conversions in PL/I source programs.

2. When the value assigned to a compile-time variable is a character string, said character string should not contain a compile-time statement nor should it contain unmatched comment or character-string delimiters. The reason for this is that such values cannot be rescanned and will therefore be considered in error when, and if, a rescan is attempted.

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. Both compile-time variables and compile-time procedure references 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; i.e., it must appear in the context of a PL/I identifier.

   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). The surrounding quctes are not inserted; blanks are inserted immediately preceding and fcllowing 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 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:

The label that determines the point at which the scan will resume must be the label of a compile-time statement.

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:] ...;
```

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 rule:

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.

THE DO GROUP

General format:

```
% [label:] ...  DO[i = m1 TO m2 [BY m3]];
          .
          .
          .
% [label:] ...  END [label];
```

Syntax rule:

The i 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 example called "Compile-Time Loop Expansion" in the section "Rescanning and Replacement" can be expressed simply as follows:

```
% DECLARE I FIXED;
% DO I = 1 TO 10;
Z(I) = X(I) + Y(I);
% END;
% DEACTIVATE I;
```

3. The semantics are the same as for source program DO groups.

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

  {identifier-1 [(identifier-2)]}
  {[identifier-1] (identifier-2)}

    ,[identifier-3 [(identifier-4)]] ...;
     [identifier-3] (identifier-4)
```

General rules:

1. Each pair of identifiers 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.

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.) An analogous situation occurs with nested DO loops; an inner loop can transfer control to an outer containing loop but not vice versa.

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(5),
    FILENAME2 FILE RECORD OUTPUT
          DIRECT KEYED(5);
```

then the program text would contain the following statement:

```
DECLARE
    MASTER  FILE  RECORD  INPUT DIRECT
       KEYED(5),
    NEWFILE FILE RECORD OUTPUT  DIRECT
       KEYED(5);
```

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 COMPILE-TIME PROCEDURE

A compile-time procedure is an internal procedure that can be executed only at the processor stage. Its syntax differs from an ordinary PL/I internal procedure 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(expression);
                    .
                    .
                    .
          % [label:]... END [label];
```

The foregoing format defines an internal function procedure that may be used at compile-time to compute a compile-time value. Each identifier is a parameter for the procedure, and the CHARACTER or FIXED attribute describes the value returned.

In addition to the RETURN statement shown in the above format, the only statements and groups that may be used within a compile-time procedure are:

1. The null statement
2. The DECLARE statement
3. The assignment statement
4. The GO TO statement
5. The IF statement
6. The DO group

The syntax and meaning of these statements and the DO group is exactly that described earlier in this chapter, the only exception being that the percent symbols must be omitted.

A compile-time procedure can be invoked by a function reference in a compile-time statement, or, if the procedure has been activated, by a reference to its name in the source program. A GO TO statement appearing within a compile-time procedure may not transfer control to a point outside that procedure.

When the Compile-Time Processor encounters a compile-time procedure during normal scanning, it notes the procedure and skips to the text immediately following the % END statement for the procedure.

The example that follows illustrates how compile-time procedures can be used.

Example: Source Program Reference to a Compile-Time Procedure

In the statements below, VALUE is a compile-time procedure that returns a value of the form arg1(arg2), where arg1 and arg2 represent the arguments that are passed to the procedure.

The source text contains the following:

```
% DECLARE    A   CHARACTER,   VALUE   ENTRY
          (CHARACTER,     FIXED)    RETURNS
          (CHARACTER);
DECLARE (Z(10),Q) FIXED;
% A = 'Z';
% VALUE: PROCEDURE (ARG1, ARG2) CHARACTER;
   DECLARE ARG1 CHARACTER,
           ARG2 FIXED;
   RETURN (ARG1||'('||ARG2||')');
% END VALUE;
Q = 6 + VALUE (A,3);
```

The last statement invokes the procedure VALUE. However, before the arguments are passed, A is replaced by its value, Z, and the character string 3 is converted to FIXED. Thus, the value returned by VALUE is the string Z(3). This value is then inserted into the program text in place of the procedure reference. The program text will therefore be as follows:

```
DECLARE (Z(10),Q) FIXED;
Q = 6 +  Z(3);
```

THE COMPILE-TIME BUILT-IN FUNCTION SUBSTR

The built-in function SUBSTR is the only built-in function that can be invoked at compile-time. It can be invoked by a reference to its name in either a source program statement or a compile-time statement.

A source program reference to SUBSTR will be executed at compile-time only if the name SUBSTR has been explicitly activated by an ACTIVATE statement. For such a reference, the arguments are treated in the same way that arguments are treated in a source program reference to a compile-time procedure (see "The DECLARE Statement" in this chapter).

A compile-time reference to SUBSTR will be executed regardless of whether or not SUBSTR has been activated (see "The Assignment Statement" in this chapter for additional information).

RELATIONSHIP OF ARGUMENTS AND PARAMETERS

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 may be a scalar, array, or structure name (including a label variable name, a task name, an array name, or an event 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 may not 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 must be declared in the invoked procedure; they cannot be declared in outer containing blocks. If no explicit declaration is given, an implicit or contextual declaration is assumed, internal to the invoked procedure.

Parameters cannot be declared with the storage class attributes STATIC, AUTOMATIC, or CONTROLLED (pointer variable) with scope attributes, or with the DEFINED attribute.

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.

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:

- a constant,
- an entry name,
- an expression involving operators,
- an expression in parentheses, or
- an expression whose data attributes may disagree with the declared data attributes of the parameter.

In all other cases the argument as it appears is passed. The parameter becomes identical with the passed argument; thus, changes to a parameter will be reflected in the original argument only if a dummy is not passed.

USE OF THE ENTRY ATTRIBUTE

An ENTRY attribute may be specified for the invoked entry name; this ENTRY attribute appears in a DECLARE statement whose scope includes the invoking block. If an ENTRY attribute is not specified in the invoking procedure for the invoked entry name, the attributes of the arguments must agree with those of the corresponding parameters of the invoked entry.

If an ENTRY attribute without parameter attribute lists is specified for an identifier, it indicates that the identifier is an entry name. In this case also, the argument and parameter attributes are assumed to agree.

However, if an ENTRY attribute with parameter attribute lists is specified for the invoked entry name, then the attributes of the parameter of the invoked entry are assumed to be the same as those specified for it in the ENTRY attribute specification. If an argument has data attributes that differ from the corresponding set of attributes defined in the ENTRY attribute specification (string lengths are considered to match only if they have the same decimal integer constant as length), then a dummy argument, with the value of the given argument, is constructed by converting the argument to the data attributes defined for the corresponding parameter in the ENTRY attribute specification. If conversion is impossible, then the program is in error

(e.g., conversion of file name to bit). The dummy argument is then passed to the invoked entry. Dummy arguments have CONTROLLED storage class in the invoking procedure. They are allocated immediately before invocation of the procedure and freed upon return, unless the invocation has a task option, in which case they are freed upon exit from the invoking block.

The asterisk notation may be used in the ENTRY attribute to specify that for varying length strings, or arrays of adjustable dimensions, the current argument bounds or length are to be assumed for the parameter.

Example:

```
A:  PROCEDURE;
    DECLARE B ENTRY (FIXED, FLOAT),
       (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 with attribute FLOAT. 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 must agree with the corresponding attributes of the parameter.

If a parameter of an invoked entry is an array, the argument must be an array expression. The argument may be a scalar expression so long as an ENTRY attribute is given for the invoked entry, specifying the dimension attribute for the relevant parameter. Asterisks may not be given in the dimension attribute if the argument is a scalar. 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 the asterisk notation is not used to specify the dimensions of the parameter in the invoked procedure, the values of the bounds of the array argument must agree with the values of the bounds specified for the parameter in the invoked procedure.

If a parameter is a structure, the argument 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.

If a parameter is a cell, the corresponding argument must be a cell whose relative structuring is the same as that of the parameter, although the level numbers need not be identical.

If a parameter is a scalar-label variable, the argument must be a scalar-label variable or constant. If a parameter is an array-label variable, the argument 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 variable or constant; 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; this information is passed as a dummy argument to the invoked entry.

If a parameter is an entry parameter, the argument must be an entry name or entry parameter. When a parameter is specified as an entry parameter in the parameter description of an ENTRY attribute and is not given data attributes, no default data attributes are assumed. If it is necessary that the entry parameter have data attributes, they may be specified in the parameter description and a check will be made

to insure that a correct argument is provided.

If a parameter is a file parameter, the argument must be a file name or file parameter. It is not necessary for the file parameter and argument attributes to match, although on use of the file parameter, the attributes of the argument must not cause any conflict with the merged attributes as specified in Chapter 7.

An argument passed to a parameter that is a fixed-length string variable or an array of fixed-length string elements must be of fixed length if no dummy argument is to be created. An argument passed to a parameter that is a varying-length string variable or an array of varying-length string elements must be of varying length if no dummy argument is to be created.

Example:

```
M1: PROCEDURE;
    DECLARE A(10), AA(10), AAA(10),
       N EXTERNAL;
            .
            .
            .
    N=10; CALL S1(A,AA,AAA);
            .
            .
            .
    END M1;

S1: PROCEDURE (P,PP,PPP);
    DECLARE P(10),PP(*),PPP(N),
       N EXTERNAL;
            .
            .
            .
    END S1;
```

In the above example, P, PP, and PPP are parameters. Procedures M1 and S1 are both external procedures. P is declared with constant bounds; thus, the bounds of any argument associated with P must be 10. PP is declared with the asterisk notation; thus, any one-dimensional argument of the same type may be associated with it. PPP is declared with an adjustable bound; thus, the bound of any argument associated with PPP must be equal to the value of N when S1 is activated. Note that a similar effect would result if S1 were internal to M1 and N were an internal variable declared in M1.

ALLOCATION OF PARAMETERS

A parameter that has no storage class 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. A nonbased 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. 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.

Parameters, Bounds and Length

If an argument is a string or an array, the length of the string or the bounds of the array must be declared in the invoked procedure by using the asterisk notation, by giving explicit bounds or length or by declaring the bounds or length as an expression that, when evaluated, gives the appropriate value. The expressions specified for the bounds or length must be formulated according to the rules stated in "Evaluation of Expressions," in Chapter 3.

The number of dimensions and the bounds of the array argument or the length of the string argument must be the same as those of the corresponding parameters. However, the actual bounds or length may not be known at the time the invoked procedure is written; the invoked procedure may assume either that storage has been allocated prior to the invocation or that storage will be allocated explicitly in the procedure for those parameters declared CONTROLLED.

Asterisk Notation for Bounds or Length

The correspondence between argument and parameter in the invoked procedure can be achieved by specifying the length by an asterisk or by specifying each and every bound by an asterisk, thus indicating that the length or bounds are the same as those for the corresponding argument.

If storage has been allocated for an argument, the corresponding parameter in the invoked procedure is assumed to have the same length or bounds as the argument. If the parameter is CONTROLLED, further allocations of the data will use these same bounds or length unless different length or bounds are specified in the ALLOCATE statement.

If storage has not been allocated for an argument passed to a parameter declared with the asterisk notation, explicit bounds or length must be declared in an ALLOCATE statement given before another reference to the parameter in the invoked procedure.

Expressions as Bounds or Length

If storage has been allocated for an argument passed to a parameter for which explicit bounds or length are specified, then upon entry to the invoked procedure, any expressions are evaluated and must give values such that the bounds or length of the parameter are the same as the argument. If the parameter is CONTROLLED and is subsequently reallocated, these expressions are again evaluated to give new bounds or length for the new allocation, unless they are specified in the ALLOCATE statement.

If storage has not been allocated for the argument, then, at the point of entry, no requirements are made on the value of the expressions specified for the corresponding parameter bounds or length. These expressions are evaluated at a subsequent point of allocation, unless they are specified in the ALLOCATE statement.

Example:

```
M2: PROCEDURE;
    DECLARE A(10), AA(25) CONTROLLED;
          .
          .
          .
    CALL S2(A,AA,10);
          .
          .
          .
    END M2;

S2: PROCEDURE (P,PP,N);
    DECLARE PP(*) CONTROLLED, P(N),
       Q(25), S(5);
          .
          .
          .
    ALLOCATE PP(25);
          .
          .
          .
    PP = Q;
          .
          .
          .
    ALLOCATE PP(5);
          .
          .
          .
    PP = S;
          .
          .
          .
    END S2;
```

DATA KNOWN TO INVOCATIONS OF RECURSIVE PROCEDURES

Each time a procedure is invoked recursively, a new generation of every automatic variable is created. If the procedure contains an internal procedure, then, within that internal procedure, the automatic data declared in the recursive procedure and known in the internal procedure is of the same generation as the internal procedure. The following examples illustrate the above discussion.

Example 1:

```
P:  PROCEDURE(Q)RECURSIVE;
     DECLARE(Q,R)ENTRY,I STATIC INITIAL(0),
           M AUTOMATIC;
     I=I+1; M=I;
    LAB: IF I<3 THEN CALL P(R);
           ELSE CALL Q; RETURN;
    R: PROCEDURE;
     PUT DATA(M);
     RETURN;
    END R;
END P;
```

In the first generation of P, when the statement labelled LAB is executed, I is less than 3; therefore, P is invoked recursively with the entry name R (that is, the first generation of internal procedure R) passed to it. In the second generation of P, M is equal to 2 while I is still less than 3. (Note that only the first generation of P initializes I to zero because I is in static storage.) Since I is less than 3, P is again invoked recursively, but this time with the second generation of the internal procedure R passed to it. In the third generation of P, I is equal to 3 and the third generation of M is equal to 3. Since I is not less than 3, ELSE CALL Q is executed. This execution invokes the procedure represented by the parameter Q, namely, the second generation of internal procedure R. Within this generation of R, the only generation of M that can possibly be known is the second. Therefore, since the second generation of M has a value of 2, the statement PUT DATA (M) causes 'M=2;' to be transmitted to the output stream.

Example 2:

```
P:  PROCEDURE RECURSIVE;
      DECLARE I STATIC INITIAL(0),
         M AUTOMATIC;
      I=I+1; M=1;
      IF I=1 THEN ON OVERFLOW PUT
         DATA (M);
      IF I=3 THEN SIGNAL OVERFLOW;
         ELSE CALL P;
      RETURN;
END P;
```

Since an ON-unit is treated as a procedure internal to the block in which it appears, the generations of data known in the ON-unit are, as in the previous example, those current at the time the ON statement for that unit is executed. In the above example, the ON statement is executed only for the first generation of P; therefore, the first generation of M is the only generation of M known within the ON-unit. Thus, after THEN SIGNAL OVERFLOW is executed, the ON-unit for that condition is executed and 'M=1;' is transmitted to the output stream.

PROLOGUES

On entering a block, certain initial actions are performed, e.g., allocation of storage for automatic variables. These initial actions constitute the prologue.

On entry to the prologue, the following items are available for computation:

1. Variables declared outside the block and known within it.
2. Variables declared STATIC and known within the block.
3. Arguments passed to the block.
4. The most recent generations of controlled variables known within 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.

In making these items available, the prologue may need to evaluate expressions defining lengths, bounds, iteration factors, and initial values. Such expressions may depend on items of 1, 2, 3 or 4. They may also be dependent on items 5 and 6 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 abnormal functions. The entry invoked with the INITIAL CALL attribute may be abnormal only in that it sets the data being initialized. The sequence in which the evaluations refer to any abnormal data is not defined.

Function calls within the evaluations must not refer to items being made available by the prologue.

DATA ALLOCATION ACROSS TASKS

The scope of an identifier declared in an attaching task may include the attached task. Thus, the WAIT statement should properly be used in the attaching task to avoid freeing storage allocated in the attaching task and used in the attached task.

An attached task has almost the same access to the attaching task's data as it would have if it were executed synchronously; however, when it is attached, only the generations of CONTROLLED variables current at the time of attachment are passed to the attached task. 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 storage that it has allocated. All storage allocated within a task is destroyed when that task is completed.

Allocation of Task and Event Names

Like variables, task names and event names have scope and storage class attributes. Storage will be allocated for task and event names in the same manner as for variables (by virtue of either an explicit or contextual declaration). If a given task is active and there is a task or event name associated with the task, then storage must not be released for the name until the task is terminated.

ABNORMALITY AND IRREDUCIBILITY

The ABNORMAL, NORMAL, IRREDUCIBLE, REDUCIBLE, USES, and SETS attributes are provided in PL/I to enable the compiler to generate optimized code.

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

1. All external function references are reducible, unless also specified in CALL statements.
2. All other procedure references are irreducible.
3. 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.

A procedure may possess varying degrees of irreducibility. A procedure is said to be "definitively irreducible" if it, or any procedures invoked by it, accesses, modifies, allocates, or frees external data or modifies, allocates, or frees arguments. In addition, an internal procedure is irreducible if it, or any procedures invoked by it, accesses, modifies, allocates, or frees any variables known in the invoking block. Such procedures are only definitively irreducible because the exact nature of their irreducibility is described by the USES and SETS attributes, thus inhibiting some, but not all, optimization in the neighborhood of a reference to the procedure (see "The USES and SETS Attributes" in Chapter 4).

However, if a procedure is "completely irreducible," all optimization of successive references must be inhibited. A procedure is completely irreducible if it, or any procedures invoked by it, does any of the following:

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

The IRREDUCIBLE attribute (described in Chapter 4) is used to describe such a procedure. It may also, of course, be used to describe a procedure that is "definitively irreducible."

When irreducibility is specified, the order of execution becomes significant. In particular, if an expression contains a reference to an irreducible function that may affect values in other parts of the expression, the value of the expression will, in general, depend upon the order in which data is accessed (see "Order of Evaluation of Expressions, " in Chapter 3).

If an IRREDUCIBLE procedure, referred to in a statement, allocates or frees controlled data that has been referred to elsewhere in the same statement, then the effect of the statement is undefined.

LIST PROCESSING

BASIC CONCEPTS

The purpose of this section is to develop the basic concepts of PL/I list processing, and to provide a simple illustration.

The description of data in PL/I, whether implicit or explicit, provides information on how to operate upon the data. If the data item is a structure or array, the description also specifies the relation among its components. However, the description of a data item does not, as a rule, have any bearing upon its location in storage. The location of a given data item is determined internally at the time the data is allocated. At the same time, a

device is established that may be thought of as a pointer and that serves to identify the data item. Thereafter, when the data item is required for the execution of object code, it is located by means of its associated pointer. In general, the pointer is not under the control of the programmer, and it is not referred to in the source program.

In list processing, however, such pointers do appear in the source program and can be manipulated to create and refer to lists of data. This is achieved by the use of the CONTROLLED storage class (see "Storage Class Attributes" in Chapter 4) and the definition of the data type pointer (see "Pointer Data" in Chapter 2). For example, consider the following declaration:

```
DECLARE P POINTER, ALPHA FLOAT CON-
   TROLLED (P);
```

This declares that P is a pointer, and that ALPHA is a floating-point variable, the location of which will be identified by P when reference is made to ALPHA. The variable ALPHA represents a new form of controlled variable known as a based variable. Unlike the nonbased form of CONTROLLED, the allocation of based variables has nothing to do with the stacking of data. Instead, it provides a device for describing the structure of data.

A based variable may be used in an extended form of the ALLOCATE statement to obtain dynamic, unstacked storage.

Example:

```
DECLARE ARRAY (100) CONTROLLED (PT)
   FIXED;
ALLOCATE ARRAY SET (PT);
```

It is assumed that (PT) has the attribute POINTER contextually by virtue of its position in the DECLARE statement. The ALLOCATE statement will reserve enough storage to contain ARRAY, and will set the pointer variable PT to the location identification that was obtained for ARRAY.

Sometimes it is convenient to allocate data items in some specific, labelled portion of storage, rather than obtaining random system storage. This storage is provided by means of the AREA attribute, which permits the programmer to identify and reserve a block of contiguous storage.

Example:

```
DECLARE TABLE AREA STATIC EXTERNAL;
ALLOCATE ARRAY IN (TABLE) SET (PT);
```

This ALLOCATE statement operates like that of the previous example, except that allocation is made into a particular block of storage named TABLE. The size of the storage block TABLE is, by default, implementation-defined; however, the programmer may override this default size specification by the use of dummy declarations with the declaration of TABLE (see "The AREA Attribute" in Chapter 4).

A based variable may be used to describe the structure of data that may exist in any storage class (STATIC, AUTOMATIC, or CONTROLLED). This is shown in the following example. The example also illustrates the use of the built-in function, ADDR, which provides programmer control of the value of the pointer P. This function returns a value of type pointer which identifies the data argument.

Example:

```
DECLARE      ALPHA  CONTROLLED (P)
                      FLOAT,
             BETA   STATIC FLOAT,
             GAMMA  AUTOMATIC FLOAT,
             OMEGA  CONTROLLED EXTERNAL
                      FLOAT;

L1:          P=ADDR (BETA);
     L1A:    ALPHA=ALPHA + 1;

L2:          P=ADDR (GAMMA);
     L2A:    ALPHA=ALPHA + 2;

             ALLOCATE OMEGA;
L3:          P=ADDR (OMEGA);
     L3A:    ALPHA=ALPHA + 3;
```

In this example, the based variable ALPHA serves as the description of BETA when P is set to the ADDR function of BETA at statement label L1. It serves as the description of GAMMA at L2, and as the description of OMEGA at L3 (after OMEGA has been allocated in the previous statement). The scope of the based variable is internal to the block in which it is declared. However, it may be used to identify external data when the associated pointer points to such data.

List processing applications may entail the use of more than one pointer to identify a given data element. Under these circumstances, other pointers (either pointers for other based variables or independently declared pointers) may be used to refer to a based variable. The symbol -> (pointer qualifier) is used for this purpose.

Example:

```
DECLARE 1 A CONTROLLED (P),
          2 B CHARACTER (30),
          2 C POINTER,
```

```
          2 D FIXED,
        Q POINTER;
        B = D;
```

The statement B = D is equivalent to

```
        P->B = P->D;
```

A reference to C implies

```
        P->C;
```

Moreover,

```
        Q  ->  A uses the pointer Q to
               identify A,
        Q -> B uses Q to identify B,
        Q -> C -> D uses Q to identify
               the pointer C,  which  is
               then  used to identify D.
```

Note that the pointers P and Q serve to identify the structure A as well as all of its components. Note also that a reference made to a based variable without the use of pointer qualification is taken as a reference to that variable with an implied qualifier; the implied qualifier being the pointer variable declared for that based variable.

The ability to "override" by means of a pointer value not declared with the based variable is valuable in examining and manipulating a complex list structure; for example, when it is desirable to examine some elements before or after the current list element position.

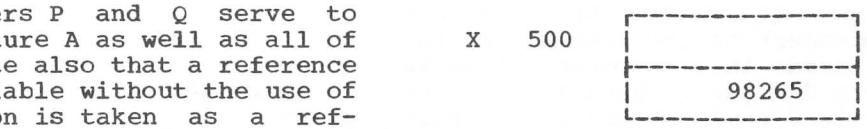
In constructing a list, a "null" pointer value is commonly used to identify the terminal entry in a list structure. This value is provided by the NULL built-in function.

With these basic definitions in mind, consider the following procedure:

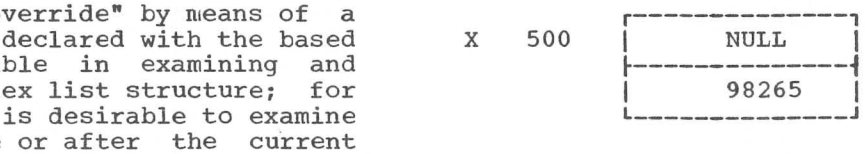
```
UNI_DIREC_CHAIN : PROCEDURE (ELEMPTR);

/*    THIS    PROCEDURE    BUILDS    A
   UNI_DIRECTIONAL DOWNWARD CHAIN THRU
   THE ELEMENTS IDENTIFIED BY THE PAR-
   AMETER ELEMPTR*/

DECLARE 1 ELEMENT CONTROLLED(ELEMPTR),
        2 P POINTER,
        2 VALUE FIXED (8,2),
    (HEAD, Q) POINTER  INITIAL  (NULL)
    STATIC EXTERNAL;

P = NULL; /* MAKE THIS ELEMENT THE NEW
    TAIL */

IF  Q  =  NULL  /*  IS  THIS THE FIRST
    ELEMENT */ THEN  HEAD,  Q  =  ADDR
    (ELEMENT); /* FIRST ELEMENT */

ELSE
Q -> P, Q = ADDR (ELEMENT); /* SUCCES-
          SOR      ELEMENT      */      END
          UNI_DIREC_CHAIN;
```

Now consider the results of two invocations of the foregoing procedure. Arbitrary location identifiers have been assigned to the arguments.

```
DECLARE 1 X STATIC INTERNAL,

   2 Y POINTER,

   2 Z FIXED (8,2);

     .
     .
     .
CALL UNI_DIREC_CHAIN (ADDR(X));
```



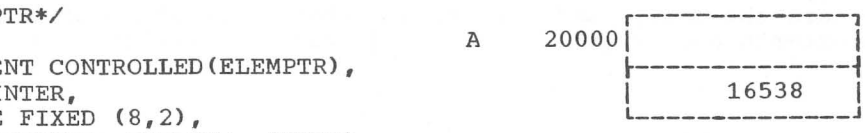
The first executable statement, P = NULL makes this element the terminal element (tail).



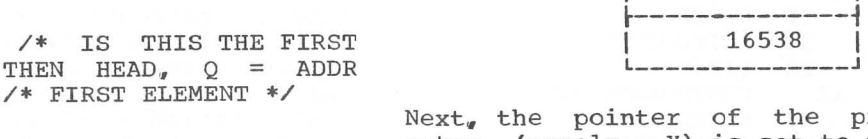
It is then determined that this is also the first element; hence,

```
            HEAD = 500
and         Q = 500
```

```
DECLARE 1 A AUTOMATIC,

   2 B POINTER,

   2 C FIXED (8,2);
     .
     .
     .
CALL UNI_DIREC_CHAIN (ADDR(A));
```

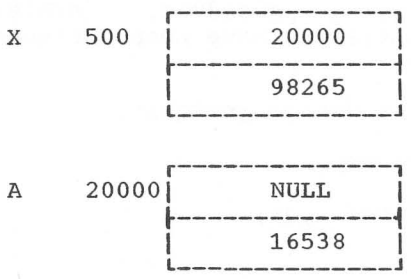


First, A is made the new tail of the list:



Next, the pointer of the previous list entry (namely, X) is set to point to A; Q is also set to point to this entry. The

parameters X, A, and Q now have the values shown below. (Note that HEAD is unchanged.)



Q = 20000

Succeeding invocations would simply add new list elements. It should be noted that there is no movement of the members of the list. They remain in the storage class in which they were declared. When members of a list reside in dynamic storage, AUTOMATIC or CONTROLLED, care should be taken to ensure that the storage is not unintentionally freed (either by leaving a block or by use of the FREE statement); such action would break the chain (see "Additional Conditions" in this chapter).

ADDITIONAL CONSIDERATIONS

Structures Used as Based Variables

When a structure appears as a based variable, it may be used to describe one or more structures in any storage class. In this case, each component of the based variable may be used to refer to the corresponding component of any of the other structures. (Two components are said to correspond if they occupy the same position relative to the beginnings of their respective structures, if the attributes of the two structures are the same, and if they have the same extents and lengths.)

Example:

```
DECLARE   1A    STATIC,
            2B  FIXED,
            2C  (10)FLOAT,
            2D  BIT(8),
          1E    AUTOMATIC,
            2F  FIXED,
            2G  (N)FLOAT,
            2H  BIT(8),
          1I    CONTROLLED (X),
            2J  FIXED,
            2K  (10)FLOAT,
            2L  BIT(8);
```

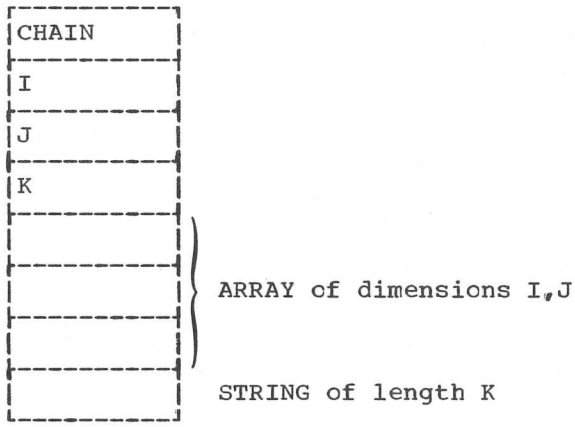
In this example, J may be used to point either to B or to F; in the first case, the pointer X would point to A, and in the second, to E. L may be used to refer to D, but it can refer to H only if the dimension of G matched that of K.

A structure which is a based variable may contain self-defining array dimensions and string lengths. This is not true of any other class of variable.

Example:

```
DECLARE 1 GROUP CONTROLLED (P),
                2 CHAIN POINTER,
                2 I FIXED,
                2 J FIXED,
                2 K FIXED,
                2 ARRAY (I,J) FLOAT,
                2 STRING CHARACTER (K);
```

This declaration takes the following form:



Assume that a pointer variable Q is pointing to a particular generation of GROUP. Any reference to Q->ARRAY will use the values of I and J contained within that generation of GROUP identified by the pointer P declared with GROUP; i.e., P->I and P->J. Similarly, a reference to STRING will use the value of K contained within that generation of GROUP identified by the pointer P declared with GROUP; i.e., P->K.

Pointer Value - Based Variable Relations

The relation between an argument and a parameter (without conversion of arguments through parameter attributes) also applies to the relation between a pointer value and the associated based variable. That is, the value of a pointer may point to a scalar variable, an array, a structure, or a component of an array or structure. The data may then be referred to by means of

the pointer value and based variable, provided the description of the based variable is compatible with that of the data identified by the pointer.

Example:

```
 DECLARE   ARRAY (10,10) STATIC EXTERNAL
           FIXED,
           VALUE CONTROLLED (P) FIXED,
           1 GROUP AUTOMATIC,
             2 GROUP_1,
               3 A FIXED,
               3 B CHARACTER (30),
             2 GROUP_2,
               3 C BIT (1),
               3 D FLOAT,
         1 DESCRIPTION CONTROLLED(Q),
               2 A FIXED,
               2 B CHARACTER (30),
       SWITCH BIT (1) CONTROLLED (R);
P = ADDR (ARRAY (I,J));
```

Provides for use of the based variable VALUE in referring to the element of ARRAY at I, J.

```
P = ADDR (GROUP_1.A);
```

Provides for the use of the based variable VALUE in referring to GROUP_1.A.

```
Q = ADDR (GROUP_1);
```

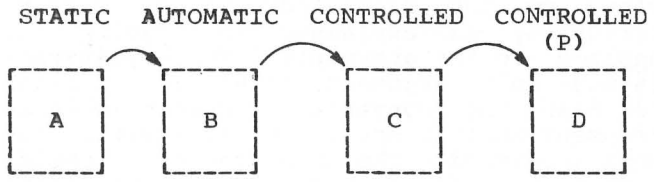
Provides for use of the based variable DESCRIPTION in referring to the minor structure GROUP_1.

```
R = ADDR (GROUP_2.C);
```

Provides for use of the based variable SWITCH in referring to GROUP_2.C.

Data Chaining Precautions

It is possible, by means of chaining, to link data which exists in different storage classes. In this case, care must be exercised to ensure that data in AUTOMATIC or CONTROLLED storage is not freed--by returning from the procedure in which the data was allocated, in the case of AUTOMATIC data, or by the execution of a FREE statement, in the case of CONTROLLED data--without adjusting the chain. Otherwise, the linkage may be lost.



If the block containing B is closed, the link between A and C is destroyed. Similarly, the freeing of C would destroy the link between B and D. The chain must, therefore, be adjusted before B or C is released.

APPENDIX 1: BUILT-IN FUNCTIONS

ARITHMETIC GENERIC FUNCTIONS

The generic functions listed in this section return a value of type coded arithmetic. The arguments may, unless otherwise specified, be any expressions. If necessary they will be converted to type coded arithmetic before the function is invoked according to the rules stated under "Type Conversion," in Chapter 3. Also certain conversions of arithmetic characteristics will be performed before the function is invoked, where this is explicitly defined to be the case for particular functions below. Where conversion to highest characteristics is specified, these are determined by the rules for mixed characteristics, as explained in Chapter 3, applied to the arguments. Where reference is made to an argument, it should be taken to mean the converted argument when an argument that is not coded arithmetic has been specified. The magnitude of a complex number is the positive square root of the sum of the squares of the real and imaginary parts where this value has the base and scale of the complex number and the mode REAL.

Name — Arguments and Function Value

ABS

Arguments: One is given.
Function value = absolute value of argument, i.e., positive value of real argument, positive magnitude of complex. The mode is REAL. Base, scale, and precision are those of the argument, unless the argument is fixed complex, in which case the precision is (MIN(N,p+1),q) for an argument of precision (p,q).

MAX

Arguments: Two or more are given. Complex arguments are not permitted.
Function value = value of maximum argument, converted to highest characteristics of all arguments specified. If the arguments are FIXED of precisions (p_1,q_1), (p_2,q_2),..., (p_n,q_n), the resulting precision is (MIN(N,MAX(p_1-q_1,...,p_n-q_n)+ MAX(q_1,...,q_n)),MAX(q_1,...,q_n))

MIN

Arguments: Two or more are given. Complex arguments are not permitted.
Function value = value of minimum argument, converted to highest characteristics of all arguments specified. If the arguments are FIXED of precisions (p_1,q_1), (p_2,q_2),..., (p_n,q_n), the resulting precision is (MIN(N,MAX(p_1-q_1,...,p_n-q_n) +MAX(q_1,...,q_n)),MAX(q_1,...,q_n)).

MOD

Arguments: two are given, x and y. Base and scale of the arguments are converted to the higher characteristics of the pair. Complex arguments are not permitted.
Function value = positive remainder after division of x by y to yield an integer quotient. The mode is REAL; base and scale are those of the converted arguments. Precision for FLOAT is the higher of the precisions of the arguments, and for FIXED is defined as follows:
Let the precision of x be (p,q) and the precision of y be (r,s). The resulting precision is (MIN(N,r-s+ MAX(q,s)),MAX(q,s)).

SIGN

Arguments: One is given. Complex arguments are not permitted.
Function value = integer 1 if argument >0; = 0 if argument = 0; = -1 if argument <0. The result is fixed binary with default precision.

FIXED

Arguments: Three are given. The second and third are optional decimal integer constants (the third may also be signed) specifying the number of digits and the scale factor of the result. If omitted, the second argument assumes a value specified by each implementation, the third assumes zero.
Function value = first argument converted to fixed-point scale with precision as specified but base and mode unchanged.

FLOAT

Arguments: Two are given. The second is an optional decimal integer constant specifying the precision of the result. If omit-

ted, a value specified by each implementation will be assumed.
Function value = first argument converted to FLOAT scale with precision as specified but base and mode unchanged.

FLOOR

Arguments: One is given, x. A complex argument is not permitted.
Function value = largest integer not exceeding x. Base, scale, and mode are those of the converted argument. Precision of result for x FIXED (p,q) is (MIN(N,MAX(p-q+1,1)),0).

CEIL

Arguments: One is given, x. A complex argument is not permitted.
Function value = smallest integer not exceeded by x. Base, scale, and mode are those of the converted argument. Precision of result for x FIXED (p,q) is (MIN(N,MAX(p-q+1,1)),0).

TRUNC

Arguments: One is given, x. A complex argument is not permitted.
Function value = FLOOR (x) if x ≥ 0, = CEIL (x) if x < 0. Base, scale and mode are those of the converted argument. Precision of result for x FIXED (p,q) is (MIN(N,MAX(p-q+1,1)),0).

BINARY

Arguments: Three are given. The second and third are optional decimal integer constants (the third may also be signed) specifying the binary precision of the result. If the scale is FIXED, and the second argument is given, the third must also be given; if the scale is FLOAT, the third is not required. If both the second and third arguments are omitted, the precision of the result is as defined for base conversion in Chapter 3.
Function value = first argument converted to binary base with scale and mode unchanged.

DECIMAL

Arguments: Three are given. The second and third are optional decimal integer constants (the third may also be signed) specifying the decimal precision of the result. If the scale is FIXED, and the second argument is given, the third must also be given; if the scale is FLOAT, the third is not required. If both the second and third arguments are omitted, the precision of the result is as defined for base conversion in Chapter 3.
Function value = first argument converted to decimal base with scale and mode unchanged.

PRECISION

Arguments: Three are given. The second and third are decimal integer constants (the third may also be signed) specifying the precision of the result. If the scale is FIXED, all three are required; if the scale is FLOAT, the third is not required.
Function value = first argument converted to specified precision. Base, scale, and mode are unchanged.

ADD

Arguments: Four are given. The third and fourth are decimal integer constants (the fourth may also be signed) specifying the precision of the result. If the scale of the result is FIXED, all four are required; if the scale is FLOAT, the fourth is not required. The third argument may not exceed N.
Function value = the sum of the first and second arguments. Base and scale of the result are the higher of those of the first two arguments. Precision is as specified.

MULTIPLY

Arguments: Four are given. The third and fourth are decimal integer constants (the fourth may also be signed) specifying the precision of the result. If the scale of the result is FIXED, all four are required; if the scale is FLOAT, the fourth is not required. The third argument may not exceed N.
Function value = the product of the first and second arguments. Base and scale of the result are the higher of those of the first two arguments. Precision is as specified.

DIVIDE

Arguments: Four are given. The third and fourth are decimal integer constants (the fourth may also be signed) specifying the precision of the result. If the scale of the result is FIXED, all four are required; if the scale

is FLOAT, the fourth is not required. The third argument may not exceed N.
Function value = the result of dividing the first argument by the second. Base and scale of the result are the higher of those of the first two arguments. Precision is as specified.

COMPLEX

Arguments: Two real arguments are given. The first is the real part, the second is the imaginary part.
Function value = complex number formed from the two arguments. Base, scale, and precision of result are the highest characteristics of those of the arguments.

REAL

Arguments: One is given, complex value.
Function value = real part of argument. Base, scale, and precision are unchanged.

IMAG

Arguments: One is given, complex value.
Function value = imaginary part of argument. Base, scale, and precision are unchanged.

CONJG

Arguments: One is given, complex value.
Function value = conjugate of the argument. Base, scale, mode, and precision are unchanged.

FLOAT ARITHMETIC GENERIC FUNCTIONS

The following generic functions may have as arguments any expression. This expression will be converted to floating point before the function is invoked. The result will be of scale FLOAT with the precision and base of the converted argument. If the mode of the argument is COMPLEX, the mode of the result will be COMPLEX. The following functions are defined only for REAL arguments: LOG2, LOG10, ATAND, TAND, SIND, COSD, ERF, ERFC, and ATAN with two arguments.

The following table specifies the meaning of these functions for real arguments:

Function Reference	Function Value
EXP (x)	exp (x)
LOG (x)	ln (x). Error if x≤0.
LOG10 (x)	$\log_{10}$ (x). Error if x≤0.
LOG2 (x)	$\log_2$(x). Error if x≤0.
ATAND (x)	arctan (x) in degrees.
ATAN (x)	arctan (x) in radians.
	ABS (arctan (x)) <pi/2.
TAND (x) degree argument	tan (x)
TAN (x) radian argument	tan (x)
SIND (x) degree argument	sin (x)
SIN (x) radian argument	sin (x)
COSD (x) degree argument	cos (x)
COS (x) radian argument	cos (x)
TANH (x) radian argument	tanh (x)
ERF (x)	Two divided by square root of pi, multiplied by the integral from 0 to x of EXP ($-t^2$) with respect to t.
SQRT (x)	The positive square root of x. Error if x<0.
ERFC (x)	1 - ERF (x)
COSH (x) radian argument	cosh (x)
SINH (x) radian argument	sinh (x)
ATANH (x)	arctanh (x). Error if AES (x)≥1.

ATAN(y,x) The arguments are converted to the highest characteristics of the pair. The value is:

arctan(y/x)	ifx>0
pi/2	if x=0, y>0
error	if x=0, y=0
-pi/2	if x=0, y<0
pi+arctan(y/x)	if x<0, y≥0
-pi+arctan(y/x)	if x<0, y<0

ATAND(y,x) ATAN(y,x) in degrees, i.e. (180/pi)*ATAN(y,x)

With complex mode many of these mathematical functions are formally multiple-valued, so the following table defines the principal values which are returned by the built-in functions. Here z = x+iy is the argument, and w = u+iv is the value.

Function Reference	Function Value
EXP(z)	exp(z)
LOG(z)	Log(z), where -pi <v≤pi. Error if z=0.
ATAN(z)	(LOG((1+z)/(1-z)))/2. Error if z= +1 or -1.
ATANH(z)	iATANH(iz). Error if z= +1i or -1i.
SIN(z)	sin(z)=sin(x)cosh(y)+ icos(x)sinh(y)
COS(z)	cos(z)=cos(x)cosh(y)- isin(x)sinh(y)
SQRT(z)	z**(1/2). Either u>0, or u=0 and v≥0.
COSH(z)	cosh(z)=cosh(x)cos(y)+ isinh(x)sin(y)
SINH(z)	sinh(z)=sinh(x)cos(y)+ icosh(x)sin(y)

STRING GENERIC FUNCTIONS

The generic functions listed in this section may be used for manipulation of strings. The arguments specified as strings may be any expression. If the argument is arithmetic, it will be converted to bit string (if binary base) or character string (if decimal base) before the function is invoked.

Name — Arguments and Function Value

BIT

Arguments: Two are given. The second is an optional decimal integer specifying the size of result.
Function value = first argument converted to type bit string. If the size is unspecified, the size of the result will be a function of the first argument characteristics (see "Type Conversion," in Chapter 3).

CHAR

Arguments: Two are given. The second is an optional decimal integer specifying the length of result.
Function value = first argument converted to type character string. If the length is unspecified, the length of the result will be a function of the first argument characteristics (see "Type Conversion," in Chapter 3).

SUBSTR

Arguments: Three are given. The first is a string, the second is any expression having the value i when converted to integer, the third is optionally any expression having the value j when converted to integer.
The function value is defined as follows:
Let k be the length of the first argument.
If i>k, the value is the null string.
If i≤k, the value is that substring beginning at the Mth character or bit of the first argument, and extending N characters or bits, where M and 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.

INDEX

Arguments: Two are given. If both arguments are bit strings, no conversion occurs, otherwise conversion to character string is performed.
Function value = binary integer of default precision giving:
a. The index of the first element of the first argument such that starting at this element the second argument appears as a substring.
b. Zero, if no such index satisfying (a) exists, or if either of the arguments is of zero length.

LENGTH

Arguments: One is given, a string.
Function value = fixed binary integer of default precision giving current length of argument.

HIGH

Arguments: One is given, a decimal integer constant.
Function value = character string of the length specified and composed of the highest characters of the data character set.

LOW

Arguments: One is given, a decimal integer constant.
Function value = character string of the length specified and composed of the lowest characters of the data character set.

REPEAT

Arguments: Two are given. The first is a string and the second is an optionally signed decimal integer constant n.

Function value = string argument concatenated with itself n times, giving a total of n+1 terms in the concatenation. If n is zero or negative, the result is the argument itself.

UNSPEC

Arguments: One is given, a scalar arithmetic, string, or pointer variable.

Function value = bit string which is the internal coded representation of the argument. The length is an implementation-defined function of the argument characteristics.

BOOL

Arguments: Three are given, bit string X, Y, and W. W is converted if necessary, to a bit string of length 4, $n^1n^2n^3n^4$. This string defines which of the 16 possible boolean functions is desired, in the manner implied below.

Function value = bit string Z where if X and Y are of different lengths, the shorter is extended with zeros, and Z is of the longer length. The following table relates the jth bit of Z to the jth bits of X and Y.

Xj	Yj	Zj
0	0	n^1
0	1	n^2
1	0	n^3
1	1	n^4

GENERIC FUNCTIONS FOR MANIPULATION OF ARRAYS

A generic function for array manipulation must have as its argument an array expression that has as its value an array of scalars. Arrays of structures are not permitted.

The following generic functions have array expression arguments and return scalar values. In the following functions x is any array expression unless otherwise specified.

Function Reference	Function Value
SUM (x)	A scalar value equal to the sum of all the elements of x. Precision, mode and base are those of argument elements. Each element of the argument is converted to arithmetic FLOAT before being summed with the previous total. The result is always in floating-point scale.)
PROD (x)	As above but product.
ALL (x)	Each element of the argument is converted to a bit string. The result is a scalar bit string whose length is equal to the length of the greatest element (in terms of length) of x. The ith bit of the result is 1 if the ith bits of all of the elements of x exist and are 1; otherwise, the ith bit of the result is zero.
ANY (x)	The result is the same as for ALL(x) except that the ith bit of the result is 1 if the ith bit of any element exists and is 1; otherwise, the ith bit of the result is zero.
POLY (a,x)	a(m:n) and x(p:q) are vectors. Result is $a(m) + \sum_{j=1}^{n-m} (a(m+j) * \prod_{i=o}^{j-1} x(p+i))$ 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). A scalar second operand x is interpreted as a vector with one element, x(1). The function result is then $\sum_{j=o}^{n-m} a(m+j) *x**j$ The characteristics of the result are the higher of those of the arguments (after conversion to arithmetic type) except for scale, which is always FLOAT.

LBOUND (x,s)	s is a scalar expression which is converted to a binary integer n, of default precision. The function value is an integer of default precision giving the current lower bound of the nth dimension of x.
HBOUND (x,s)	As above but higher bound.
DIM (x,s)	s is as above. The function value is a binary integer n of default precision giving the current extent of the nth dimension of x.

NOTE: The functions LBOUND, HBOUND, and DIM are not defined if the argument x is unallocated, if it has less than n dimensions, or if n≤0.

ARRAY AND STRUCTURE BUILT-IN FUNCTIONS

All of the built-in functions listed under "Arithmetic Generic Functions" and "String Generic Functions" in this appendix may have array or structure expressions as arguments, except where decimal integer constants are required. They yield an array or structure of the same dimension bounds or structuring as the argument--the function being performed on each element. The rules are the same as those for the scalar functions.

CONDITION BUILT-IN FUNCTIONS

The following built-in functions (with no arguments) are available to allow investigation of interrupts arising from enabled ON conditions.

Function Reference	Function Value
ONFILE	A character string of varying length with an implementation-defined maximum, being the name of the file for which the last input/output operation was performed. If there is no such file, the value returned is the null string.
ONLOC	A character string of varying length with an implementation defined maximum length, being the entry point name of the immediate dynamically encompassing procedure in which the last interrupt arose.
ONSOURCE	A character string of varying length, with an implementation defined maximum length, being the contents of the field being processed when the last conversion interrupt occurred.
ONCHAR	A character string of length 1, being the character which caused the last conversion interrupt.
ONKEY	A character string of varying length, with an implementation defined maximum length, being the value of the key for the record whose transmission caused the last interrupt.
ONCODE	A binary integer of default precision whose value specifies the last interrupt. The categories and code for each are implementation-defined.
DATAFIELD	A character string of varying length, with an implementation defined maximum length, being the contents of the data field which gave rise to the last NAME condition interrupt.

LIST PROCESSING BUILT-IN FUNCTIONS

The following functions are used in connection with list processing to provide suitable values of type pointer.

Function Reference	Function Value
ADDR (x)	The function returns a value of type pointer, which serves to identify the data variable x. The variable x may be a scalar, an array, a structure, an element of an array, or a component of a structure. If x is a formal parameter, the value is determined from the corresponding argument. If x is a nonbased controlled variable, the value is determined from the most recent generation (see "The ALLOCATE Statement" in Chapter 8); if x is unallocated, the value is NULL. If x is a based variable, the value is determined from the pointer variable declared

with it; if the pointer variable does not contain a value, the ADDR function value is not predictable.

NULL — This function defines a null pointer value; hence, it does not identify any generation of data. Its value is implementation defined.

OTHER BUILT-IN FUNCTIONS

Function Reference	Function Value
DATE	Character string of length six of the form YYMMDD, where YY is year, MM is month, DD is day.
TIME	Character string of length nine of the form HHMMSSTTT, where HH is hours, MM is minutes, SS is seconds, TTT is milliseconds.
ALLOCATION (x)	x is a nonbased CONTROLLED major structure or unsubscripted array or scalar variable not in a structure. The function value is '1'B if storage has been allocated for x and '0'B if not.
LINENO (filename)	The value of this function is a binary fixed-point integer of default precision. It specifies the current line number of the specified PRINT file.
COUNT (filename)	The value of this function is a binary fixed-point integer of default precision. It returns a value that is the number of scalar data items transmitted during the last GET or PUT operation on the specified file.
ROUND (expression, decimal-integer-constant)	The expression may be a scalar, array, or structure. The decimal integer constant (call it n) may be signed. If n is positive, the value returned by the function is the expression value rounded on the nth digit to the right of the decimal point. If n is negative, the value of the function is an integer resulting from rounding the expression value on the nth digit to the left of the decimal point. (Binary digits if binary base, decimal if decimal base.) If the expression is of string type, the function value is the string value unmodified. Floating point rounding is a bias removal rather than systematic rounding; the decimal point is assumed at the left. Base, scale, mode and precision of the value are those of argument. If the scale is FIXED with precision (p,q), the result is FIXED with precision (MIN(p+1,N),q). (Note that the rounding of a negative fixed-point quantity results in the rounding of the magnitude of that quantity.)
STRING (structure-name)	The argument must be a packed structure composed either of all bit strings and numeric fields of binary base, or character strings and numeric field of decimal base. The function value is a string, being the concatenation of all the structure elements.
EVENT (scalar-event-name)	This function will return the value '0'B OR '1'B, depending on the current status of the referenced event name (see "Asynchronous Operations and Tasks," in Chapter 6 and "The WAIT Statement," in Chapter 8).
PRIORITY (scalar-task-name)	This function will return the priority of the named task relative to the priority of the task in which the function is evaluated (see "Asynchronous Operations and Tasks," in Chapter 6 and "The WAIT Statement," in Chapter 8).

DIGIT POINT AND SUBFIELD DELIMITING CHARACTERS

9 Specifies that the associated field position will contain any decimal digit.

1 Specifies that the associated field position contains a binary digit. This character may not appear in a picture with either 2 or 3.

2 Specifies that the associated field position contains a binary digit, being part of a binary value in 2's complement notation. This character may not appear in a picture with either 1, 3, or S.

3 Specifies that the associated field position contains a binary digit, being part of a binary value in 1's complement notation. This character may not appear in a picture with either 1, 2, or S.

V Specifies that a decimal or binary point should be assumed to appear at this point in the associated field. It does not specify a character in the field.

K Specifies that the exponent subfield should be assumed to follow the point in the field associated with the K. It does not specify a character in the field.

E Specifies that the associated field position will contain the letter E, indicating the start of the exponent subfield.

ZERO SUPPRESSION CHARACTERS

A leading zero in a numeric subfield is a zero to the left of the actual occurrence of the digits 1 to 9 in the subfield. The leftmost of these latter digits and all digits in the subfield following it, are significant digits (including any zeros). Picture characters are provided for zero suppression, leading zero suppression, and the replacement of these zeros by blanks or asterisks.

Z Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by a blank, otherwise the digit will appear. The character may not appear to the right of 9 T I R or a drifting string in a subfield. It may not appear with * in a subfield.

* Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by *, otherwise the digits will appear. The character may not appear to the right of 9 T I R or drifting string in a subfield. It may not appear with Z in a subfield.

Y Specifies a conditional digit position. If the associated field position involves a zero (leading or otherwise) it will be represented in the field by a blank, if it involves a digit other than zero that digit will appear.

DRIFTING EDITING SYMBOLS

The following picture characters may be static or drifting:

Character	Name
S, +, -	sign characters
$	currency symbol

The static use of these characters specifies that there is a field position where a sign, a currency symbol, or a blank always appears. The drifting use specifies that leading zeros may be suppressed, and the suppressed positions may contain blanks. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a dollar sign.

A drifting character is specified by multiple use of that character in a picture subfield. Thus, if a subfield contains one dollar sign, it is interpreted as static; if it contains more than one, as drifting. The drifting character must be specified in each position through which it may drift.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing interspersed editing characters comma (,), point (.), slash (/), or V or B. Picture characters slash, comma, point, and B following the last drifting symbol of the string are considered part of the string. However, a following V terminates the string and is not part of it. A subfield

may only contain one drifting string. The picture characters * and Z may not appear to the right of a drifting string in a subfield.

The field position associated with the character slash, comma, point, and B appearing in a drifting string will contain one of the following:

1. Slash, comma, point, or blank if a significant digit has appeared to the left.

2. The drifting symbol, if the next position to the right contains the leftmost significant digit of the subfield.

3. Blank, if the leftmost significant digit of the subfield 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 field position associated with the leftmost drifting character may only contain the drifting character or blank, never a digit. If a drifting string is specified for a subfield, the other potentially drifting characters may only appear once to the left of the string in the subfield, i.e., the other characters represent a static sign or dollar sign.

If a drifting string contains a V, then all digit positions of the subfield following the V must also be part of the drifting string.

If one of the characters Z or * follows the V in a subfield, then all digit positions in the subfield following the V must be Z or asterisk (*).

In the case where all digit positions after the V contain suppression characters, suppression will only occur where all the fraction digits are zero. The resulting field will then be all blanks or asterisks. If there are any significant fraction digits they all will appear unsuppressed.

DRIFTING CHARACTERS

$ If this character appears more than once in a subfield it is a drifting character, otherwise it is a static character. The static character specifies that the character $ be placed in the associated field position. The static character must appear either to the left of all digit positions in a subfield or to the right of all digit positions in a subfield. See details above for the drifting use of the character.

S Specifies the sign character + if the field value is ≥0, otherwise -. The character may be drifting or static. The rules are identical to those for the dollar sign.

+ Specifies the sign character + if the field value is ≥ to 0, otherwise blank. The character may be drifting or static. The rules are identical to those for the dollar sign.

- Specifies the sign character - if field value is <0, otherwise blank. The character may be drifting or static. The rules are identical to those for the dollar sign.

EDITING CHARACTER

B Specifies that a blank appear in the associated field position.

CONDITIONAL EDITING CHARACTERS

, If the subfields in which the comma appears involve no zero suppression, that character specifies that a comma will appear in the associated field position. If zero suppression is involved the comma will appear only if there is an unsuppressed digit to the left of the comma position in the subfield. If there is no such unsuppressed digit, the associated field position will contain a character that depends on the first digit (conditional or otherwise) picture character preceding the comma.

If the preceding character is an asterisk the field position will contain an asterisk.

If the preceding character is a drifting sign or dollar sign the action taken will be identical to that which would have occurred if the picture specification had contained the drifting character in place of the comma.

If the preceding picture character is anything other than the above, the field position associated with the comma will contain a blank.

/ Exactly as comma, but a slash will appear when indicated.

. Exactly as comma, but a point will appear when indicated.

SIGN CHARACTERS

Digit characters in numeric fields may contain an overpunched sign. The following picture characters are used to specify overpunching:

T Specifies that the associated field position will contain a digit overpunched with the sign of the containing subfield.

I Specifies that the associated field position will contain a digit overpunched with + if the containing subfield is ≥ 0; otherwise it will contain the digit with no overpunching.

R Specifies that the associated field position will contain a digit overpunched with - if the containing subfield is < 0; otherwise it will contain the digit with no overpunching.

The above characters may not be used in conjunction with any other sign characters in the same subfield.

The two character picture items CR and DB may be used to reflect the sign of REAL numeric fields.

CR Specifies that the associated field positions will contain the letters CR if the containing field value is <0. Otherwise the positions will contain two blanks. The characters CR may appear only to the right of all digit positions of a field.

DB As CR, except that a DB appears.

SCALING FACTOR SPECIFICATION

F Specifies that the optionally signed decimal integer enclosed in parentheses following the picture character F in the picture string is the scaling factor (see "The PICTURE Attribute," in Chapter 4).

STERLING PICTURES

The following additional characters are provided for use in sterling pictures.

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 field position contains the pence character D.

G Specifies the start of a sterling picture. It does not specify a character in the numeric field.

H Specifies that the associated field position contains the shilling character S.

M Specifies the start of a subfield. It does not specify a character in the numeric field.

PICTURES FOR CHARACTER STRINGS

A form of picture may be given for character strings. The following are used to indicate the form:

A The associated field position may contain any alphabetic character or blank.

X The associated field position may contain any character.

9 The associated field position may contain any decimal digit or blank.

At least one X or A must appear in the picture.

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, 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.

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 length of an arithmetic subfield is limited by an implementation restriction.

Result: 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: Truncation on the left to size N.

Standard System Action: Comment and continue.

OVERFLOW: This condition occurs when the exponent of a floating-point number exceeds the permitted maximum, as defined by the implementation.

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: 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 of the item of data receiving a value.

Result: The result is 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).

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: 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.

In some implementations, the condition may be detected by hardware interrupt, in others by special coding.

Result: 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. The condition established refers to the file value, and not necessarily to all files having a common identifier (e.g., file parameters). It is not possible to override a condition by a new setting in the same block, using a different identifier to refer to it.

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.

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. The current line becomes one more than the expression specified with the PAGESIZE option. The condition can be raised by data transmission (with associated format items--if edit-directed transmission), the LINE option, or the SKIP option. It is raised only once per page.

If raised by 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 raised by LINE or SKIP, on return, the action specified by LINE or SKIP is ignored.

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.

Standard System Action: Comment and raise the ERROR condition.

UNDEFINEDFILE (filename): This condition is raised on any OPEN statement if the named file cannot be opened. An attempt will have been made to open all other files referred to in the same OPEN statement. On return from the on-unit, processing will continue with the next statement. If this condition is raised for more than one file in the same OPEN statement, on-units will be executed according to the left-to-right order of appearance of the filenames in that OPEN statement.

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

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).

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.

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 may not contain based variables, formal 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, may have been changed 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 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 a pseudo-array, pseudo-structure, or pseudo-variable appearing on the left hand side of an assignment.

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 a GET statement.

5. ID is altered by data-directed input.

6. ID appears as the second argument of a DISPLAY statement.

7. ID appears as a STRING option of a PUT statement.

8. ID is passed as an argument to a programmer-defined procedure, no dummy is created, and the procedure terminates with a RETURN.

9. ID appears in the SET option of an ALLOCATE, READ, or LOCATE 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 the value of a variable upon which ID is defined changes value.

3. If a parameter which represents ID changes value.

4. If ID appears in a GO TO or RETURN statement or any statement which involves the execution of a GO TO or RETURN statement.

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.

No statement other than a DO statement can cause a condition to be raised more than once for the same identifier. 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 Labels: For an entry label, the condition is raised prior to each invocation of the entry label. The condition is raised only if the entry label is invoked by the name given in the ON list.

Result: Continue. The statement is executed normally.

Standard System Action: If the identifier is a statement label or an entry name, the identifier will be printed on a debugging file. Label variables, TASK, and EVENT names are treated in the same manner.

If the identifier represents data other than that mentioned above, the identifier and its new value will be 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.

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 may not appear in a condition prefix.

FINISH: This condition is raised immediately before the main procedure terminates by executing a STOP, RETURN, END, or EXIT statement. If an ON-unit for this condition is specified, it is executed as part of the task in which the interrupt takes place. Upon normal completion of the ON-unit, the system terminates the major task.

Standard System Action: Terminate the major task.

ERROR: This condition is raised when a major task is forced to terminate because of some error situation. If an ON-unit is specified for this condition, then upon normal completion of this unit, the system raises the FINISH condition.

Standard System Action: Raise the FINISH condition.

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
AUTOMATIC	AUTO
BINARY	BIN
CHARACTER	CHAR
COMPLEX	CPLX
CONTROLLED	CTL
CONVERSION	CONV
DECIMAL	DEC
DECLARE	DCL
DEFINED	DEF
ENVIRONMENT	ENV
EXTERNAL	EXT
FIXEDOVERFLOW	FOFL
INITIAL	INIT
INTERNAL	INT
IRREDUCIBLE	IRRED
OVERFLOW	OFL
PICTURE	PIC
POINTER	PTR
POSITION	POS
PRECISION	PREC
PROCEDURE	PROC
REDUCIBLE	RED
SUBSCRIPTRANGE	SUBRG
UNDERFLOW	UFL
UNDEFINEDFILE	UNDF
VARYING	VAR
ZERODIVIDE	ZDIV

APPENDIX 5: THE 48-CHARACTER SET

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.

```
 1  UPDATE:  PROCEDURE;
 2           DECLARE CHANGE FILE SEQUENTIAL
                      UNBUFFERED RECORD,
 3                   MASTER FILE INPUT BUFFERED
                      KEYED (10),
 4                   NEW_MASTER FILE BUFFERED
                      KEYED (10),
 5                   1 CHANGE_REC,
 6                     2 CHANGE_KEY CHARACTER
                        (10),
 7                     2 CHANGE_INFO CHARACTER
                        (50),
 8                   MASTER_KEY CHARACTER (10),
 9                   MASTER_INFO CHARACTER (50)
                      CONTROLLED (IN_IDENT),
10                   REC_TEMP CHARACTER (50),
11                   STATUS BIT(1) INITIAL('0'B);
12           ON ENDFILE (CHANGE) BEGIN;
13                               IF STATUS = '1'B
14                                THEN GO TO FINISH;
15                                       STATUS = '1'B;
16                                       CHANGE_KEY = HIGH (10);
17                                       END;
18           ON ENDFILE (MASTER) BEGIN;
19                               IF STATUS = '1'B
20                                THEN GO TO FINISH;
21                                       STATUS = '1'B;
22                                       MASTER_KEY = HIGH (10);
23                                       END;
24           OPEN FILE (CHANGE) INPUT;
25  L1:      READ FILE (CHANGE) INTO (CHANGE_REC);
26  L2:      READ FILE (MASTER) SET (IN_IDENT) KEYTO (MASTER_KEY);
27  L3:  IF CHANGE_KEY = MASTER_KEY
28           THEN DO;
29                   MASTER_INFO = CHANGE_INFO;
30                   WRITE FILE (NEW_MASTER) FROM
                      (MASTER_INFO) KEYFROM (MASTER_KEY);
31                   GO TO L1;
32                   END;
33           IF MASTER_KEY<CHANGE_KEY
34              THEN DO;
35                   WRITE FILE (NEW_MASTER) FROM
                      (MASTER_INFO) KEYFROM (MASTER_KEY);
36                   GO TO L2;
37                   END;
38  /* MASTER_KEY>CHANGE_KEY*/
39           REC_TEMP = CHANGE_INFO;
40           WRITE FILE (NEW_MASTER) FROM
                 (REC_TEMP) KEYFROM (CHANGE_KEY);
41           READ FILE (CHANGE) INTO (CHANGE_REC);
42           GO TO L3;
43  FINISH:  CLOSE FILE (CHANGE), FILE (MASTER), FILE (NEW_MASTER);
44                   STOP;
45           END UPDATE;
```

Example 1. An Update Procedure (line numbers are for reference only, and are not a part of the program).

Example 1 is a simple update procedure to create a new master file from an existing master file, making changes to existing records and adding new records to the file.

In line 2, the identifier CHANGE is declared to be a filename associated with a sequentially organized data set. All of the attributes, except for the function attribute, are declared explicitly in the DECLARE statement.

In line 3, MASTER is declared to have the INPUT, BUFFERED, and KEYED attributes; the key of each record is 10 characters in length. The RECORD and SEQUENTIAL attributes can be assumed, because the BUFFERED attribute is explicitly declared.

The RECORD and SEQUENTIAL attributes can be assumed for NEW_MASTER (line 4) since BUFFERED is declared. No function attribute can be assumed.

The major structure CHANGE_REC is declared in lines 5-7, with the elements CHANGE_KEY and CHANGE_INFO. The key of the update record will be read into CHANGE_KEY, and the update information into CHANGE_INFO.

MASTER_KEY (line 8) is a character-string variable into which the key from records in MASTER can be read for comparison with keys from records in CHANGE_REC.

MASTER_INFO (line 9) is a based variable that describes the record in a buffer. The CONTROLLED attribute specification contextually declares the pointer variable IN_IDENT, which can be used to specify the position of the data in the buffer.

REC_TEMP (line 10) is used, during execution of the program (lines 39 and 40), as a temporary area from which data can be written.

STATUS (line 11) is a program switch that is initialized with the bit constant '0'.

All of the files declared have the default scope attribute of EXTERNAL; all of the variables have the default scope attribute of INTERNAL and, with the exception of MASTER_INFO, the AUTOMATIC storage class attribute.

In lines 12 through 23, ON ENDFILE statements establish on-units for the end-of-file condition for CHANGE and MASTER files. Their execution is discussed below.

The OPEN statement (line 24) opens CHANGE file and explicitly adds the INPUT attribute to the filename CHANGE.

The READ statement (line 25) transfers the record from the CHANGE data set directly to the structure CHANGE_REC. There is no data conversion; the assumption is that the first 10 characters of the record represent the key and the next 50 characters represent the update information.

The READ statement in line 26 first causes implicit opening of MASTER file. It then reads the record into a buffer and sets the pointer variable IN_IDENT to point to the record in the buffer. In effect, MASTER_INFO is allocated and assigned. Consequently, any reference to the based variable MASTER_INFO is a reference to the record in the buffer. The READ statement also transfers the key of the record to the character-string-variable MASTER_KEY.

The key of the record read from MASTER is compared with the key of the record read from CHANGE (line 27). If they are the same, indicating that the MASTER record is to be updated, the update information replaces the old record in the buffer (line 29), and the updated record is written in the NEW_MASTER data set (line 30).

The NEW_MASTER file is not explicitly opened, but the first execution of a WRITE statement that refers to NEW_MASTER will cause implicit opening of the file and will contextually supply the OUTPUT function attribute to NEW_MASTER. The file opening could be caused by any of the WRITE statements (lines 30, 35, 40), depending upon which is executed first.

If the keys of the two records agree, control is returned (line 31) to the first READ statement, and two new records are read. If the keys indicate that the update record does not refer to the current record in MASTER, a test must be made to determine if the CHANGE record refers to a later MASTER record or if the CHANGE record actually must create a new record in the NEW_MASTER data set. If MASTER_KEY is less than CHANGE_KEY (line 33), it indicates there is no change to be made to the current MASTER record, and it is written (line 35) from the buffer into NEW_MASTER exactly as it was read from MASTER. Control is then returned (line 36) to read a new record from the MASTER data set.

If neither of the two IF statements (lines 27 and 33) is true, MASTER_KEY must be greater than CHANGE_KEY, which indicates that the CHANGE record is to be added to the data set in NEW_MASTER. CHANGE_INFO is assigned to REC_TEMP and the record is written in NEW_MASTER (line 40).

After the new record is written, another record is read from the CHANGE data set,

and control is transferred (line 42) back to the first IF statement.

When the first end-of-file condition is raised, the program switch STATUS is set in the on-unit (line 15 or 21), and the appropriate variable (CHANGE_KEY or MASTER_KEY) is changed so that it always will compare high in subsequent IF statements. This is accomplished (line 16 or 22) through use of the HIGH built-in function which returns a character string (in this case, of length 10) of the highest characters in the collating sequence.

When the second end-of-file condition is raised, the test of STATUS (line 13 or 19) results in a transfer to FINISH (line 43) and all files are explicitly closed.

```
 1  LIST:    PROCEDURE (AUTHOR, NUMBER_PUBS);
 2           DECLARE AUTHOR CHARACTER (30),
 3                     PUBLICATIONS FILE DIRECT INTERNAL KEYED(30),
 4                     LISTING FILE STREAM PRINT,
 5                     FIRST_TIME BIT (1)INITIAL ('0'B) STATIC,
 6                     AUTHOR_PUBS (NUMBER_PUBS) AUTOMATIC CHARACTER (100);
 7           IF FIRST_TIME = '0'B
 8             THEN DO;
 9                  OPEN FILE (LISTING) LINESIZE (120) PAGESIZE (58);
10                  PUT FILE (LISTING) EDIT ('AUTHOR PUBLICATIONS') (COLUMN (5), A);
11                  PUT FILE (LISTING) LINE (2);
12                  FIRST_TIME = '1'B;
13                  END;
14           ON ENDPAGE (LISTING) BEGIN;
15                               PUT FILE (LISTING) EDIT
                                 ('CONTINUED ON NEXT PAGE',
                                 'AUTHOR PUBLICATIONS CONTINUED')
                                 (SKIP, A, PAGE, COLUMN (5), A);
16                               PUT FILE (LISTING) SKIP;
17                               END;
18           IF AUTHOR = (30)''
19             THEN DO;
20                  PUT FILE (LISTING) EDIT ('END OF AUTHOR INDEX') (SKIP (2), A);
21                  CLOSE FILE (PUBLICATIONS), FILE (LISTING);
22                  RETURN;
23                  END;
24                  PUT FILE (LISTING) EDIT (AUTHOR)
                      (SKIP, COLUMN (10),A);
25                  IF NUMBER_PUBS>0
26                    THEN DO;
27                           READ FILE (PUBLICATIONS) KEY
                                   (AUTHOR) INTO (AUTHOR_PUBS);
28                           PUT FILE (LISTING) EDIT (AUTHOR_PUBS)
                                   (R(PUB));
29                           RETURN;
30                           END;
31           PUT FILE (LISTING) EDIT ('NO PUBLICATIONS') (R(PUB));
32  PUB:     FORMAT (SKIP,COLUMN(15),A);
33           END LIST;
```

Example 2. An Information Retrieval and Listing Procedure (line numbers are for reference only, and are not part of the program).

Example 2 is a simple information retrieval and listing procedure. It extracts information from a file of PUBLICATIONS, based upon requests indicated by the parameters AUTHOR and NUMBER_PUBS passed to the procedure when it is invoked. The information subsequently is printed in a LISTING file.

The declaration of the parameter AUTHOR (line 2) indicates it is a character string of length 30. This parameter is used as a key for locating publication information in the PUBLICATIONS data set. Note that the parameter NUMBER_PUBS is implicitly declared with the FIXED, BINARY, and REAL attributes.

The PUBLICATIONS file is declared (line 3) to be a DIRECT file, with 30-character keys that represent the authors' names. The file is declared to have the INTERNAL scope attribute; the RECORD attribute is implied from the other attributes. The file LISTING is explicitly declared (line 4) with the STREAM and PRINT attributes, with PRINT implying OUTPUT.

FIRST_TIME is declared (line 5) as a program switch used to control initial actions in the procedure.

AUTHOR_PUBS (line 6) is a one-dimensional character-string array into which the list of publications is read and from which the list is printed. Since it has the AUTOMATIC storage class attribute, it is allocated each time the procedure is invoked, with the number of elements depending upon the current value of NUMBER_PUBS, which is the number of publications by the author named.

The IF statement (line 7) tests the switch FIRST_TIME to determine its value. The THEN clause (lines 8 through 13) will be executed only once, the first time the procedure is invoked. Since FIRST_TIME has the STATIC attribute, its setting will remain even after the procedure is terminated at the end of each execution.

The THEN clause includes the opening of the LISTING file (line 9), which sets the length of lines and the number of lines to be printed on each page of the listing. The initial heading is written (line 10), with an indentation of five characters; then the PUT statement (line 11) makes the current line become line two. Following that, FIRST_TIME is set (line 12) so that the THEN clause will be skipped in subsequent executions of the procedure.

The ON ENDPAGE statement (line 14) must be executed each time the procedure is invoked to reestablish the on-unit (lines 15 through 17). The on-unit provides for printing a footing, 'CONTINUED ON NEXT PAGE' at line 60 of the page (the ENDPAGE condition arises when the current line is at PAGESIZE + 1, and the SKIP format item provides another skip). Following printing of the footing, the PAGE format item causes creation of a new current page, and the heading 'AUTHOR PUBLICATIONS CONTINUED' is written with an indentation of five characters. The next PUT statement (line 16) causes skipping of another line.

The IF AUTHOR = (30) '' statement (line 18) is a test of a convention of the program: when the listing is complete, the invoking procedure calls LIST and passes an argument consisting of 30 blank characters to the parameter AUTHOR. At that point, the 'END OF AUTHOR INDEX' character string is printed, after skipping two lines, and the PUBLICATIONS and LISTING files are closed (line 21).

If there is a listing to be printed, the PUT statement (line 24) prints the name of the author with an indentation of 10 characters. If there are publications to be listed (determined by the IF statement in line 25), the list of publications is read from the PUBLICATIONS data set (line 27), using the author's name as a key. The data is read into the array AUTHOR_PUBS.

The publications then are printed (line 28) using the remote format item that refers to the FORMAT statement (line 32). The format items in the FORMAT statement specify that a line is to be skipped, and each publication (each element of the array) is to be printed with an indentation of 15 characters.

When printing is complete, control is returned to the invoking procedure (line 29).

If the author's name is to be written, but with no publications, the DO group is skipped, and 'NO PUBLICATIONS' is printed where the first publication would otherwise have been printed.

(If more than one page number is given, the primary discussion is listed first.)

Printed in U.S.A.

C28-6571-3

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601

READER'S COMMENTS

Title: IBM System/360 Operating System
PL/I Language Specifications

Form: C28-6571-3

Is the material:	Yes	No
Easy to read?	---	---
Well organized?	---	---
Complete?	---	---
Well illustrated?	---	---
Accurate?	---	---
Written for your technical level?	---	---

How did you use this publication?

-----As an introduction to the subject -----For additional knowledge

Other----------

Please check the items that describe your position:

-----Customer personnel ----Operator -----Sales Representative
-----IBM personnel ----Programmer -----Systems Engineer
-----Manager ----Customer Engineer -----Trainee
-----Systems Analyst ----Instructor -----Other----------

Please check specific criticisms, give page numbers, and explain below:

-----Clarification on pages
-----Addition on pages
-----Deletion on pages
-----Error on pages

Explanation:

If you wish a reply, be sure to include your name and address.

fold fold

FIRST CLASS
PERMIT NO. 33504
NEW YORK, N.Y.

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM CORPORATION

1271 AVENUE OF THE AMERICAS
NEW YORK, N.Y. 10020

ATTENTION: PUBLICATIONS, DEPT. D39

fold fold

C28-6571-3

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601